Facultade de Informática
UNIVERSIDADE DA CORUÑA

TRABALLO FIN DE GRAO
GRAO EN ENXEÑARÍA INFORMÁTICA
MENCIÓN EN ENXEÑARÍA DO SOFTWARE

Project management platform for the identification of phytoplankton samples

Estudante: Francisco Bretal Ageitos
Dirección: José Rouco Maseda
Jorge Novo Buján

A Coruña, setembro de 2020 .

To this memorable journey

Acknowledgements

I would not imagine writing what is my first dissertation without the outstanding effort that my parents have made to provide me with the tools and opportunities to study all these years, so it feels right to say how thankful I am of having them.

I would also want to thank all my friends, who helped me throughout this life's journey, that really made me become the person that I am today.

Thanks to the university and all the people that I met these years, that let me grow as a person and also gave me some of the best experiences one could ask for.

Finally, but not least, I would like to thank both of my dissertation directors for trusting me with this project, and for giving me the freedom to imagine it as my own.

Abstract

Currently, experts on the field of biology are tasked with the analysis of multiple samples of water masses to ensure their potability. All of this work is performed manually due to the lack of fitting alternatives in the state of the art.

It was necessary to develop a new platform aimed at providing a way to centralize all this work. The platform's mission is to bring carefully designed tools to fit the identification process, while releasing scientists from some workload.

Furthermore, automation has a crucial role in the future of the field, and the platform should be able to support it as well. Thus, providing both manual identification of phytoplankton by human experts and automated identification.

Resumo

Na actualide, expertos no ámbito da bioloxía están encargados da análise de mostras de diversas masas de auga para asegurar a súa potabilidade. Todo este traballo faise de maneira individual debido ás carencias do estado da arte.

É necesario desenvolver unha nova plataforma coa que se pretende proporcionar unha maneira de centralizar todo este traballo. O obxectivo da plataforma é o de traer novas ferramentas deseñadas para axustarse ás necesidades do proceso de identificación, ó mesmo tempo que os científicos obterían certas facilidades nos seus cometidos.

Además, a automatización ten un papel fundamental no futuro do campo, polo que a plataforma deberá estar preparada para soportala. De esta maneira, a plataforma soportará tanto a identificación manual dos expertos como identificacións automáticas feitas por máquinas.

Keywords:

- Phytoplankton
- Identification
- Tagging
- Web application
- MVC
- Laravel
- PHP

Palabras chave:

- Fitoplancton
- Identificación
- Etiquetado
- Aplicación Web
- MVC
- Laravel
- PHP

Contents

List of Figures

List of Tables

Chapter 1
Introduction

We are currently living in the digital era, where we benefit from a substantial array of technologies that help us withstand life's many obstacles with ease. From minuscule computers in our pockets that were unthinkable just a few decades ago, to automated vacuum that intelligently maintain our homes clean, we encounter ourselves surrounded by technological advances.

Unbeknownst to many people, however, technology plays a crucial role behind the scenes, indirectly affecting our quality of life by acting as a foundation for new scientific discoveries.

1.1 Motivation

Something just as common, and oftentimes unappreciated, as is running water can experience a lot of complex processes before reaching our kitchen's tap. All those processes ensure that we do not have to think about anything else but drinking it.

To guarantee the potability of reservoirs used as supply of running water, biologists are in charge of periodically analyzing them beforehand. However, their work does not end there: lakes, rivers and even beaches have to be continuously surveyed for malicious species that can bloom at any moment if left unsupervised.

With their work, they are able to control the population of microscopic organisms that might produce toxins that, when reaching a certain threshold, can make water very harmful for humans to consume. Among these dangerous creatures, phytoplankton resides.

Phytoplankton is a group of microscopic species found in water. They can synthesize on their own all that is required for them to normally function, and are also able to photosynthesize [1], hence, their name.

A subset of this family of species have the ability to produce harmful toxins, but not all of them. It is the scientists' task to single out the hazardous species found in water samples in order for them to be able to report the potability of the source.

If that were not enough, biologists also participate in numerous studies regarding water analysis. All of this very methodical work can become very cumbersome and inefficient without the right tools to support it.

1.2 State of the Art

The current State of the Art involves scientists identifying species in water samples manually. This is understandable when there are not many alternatives out there to help with their work.

Some platforms out there focus their work on relieving the scientist from the manual, most repetitive work: the identification itself. If we look at platforms like PlanktonID [1], they are using an approach where anonymous users from the internet are asked to help in the identification process, as a game.

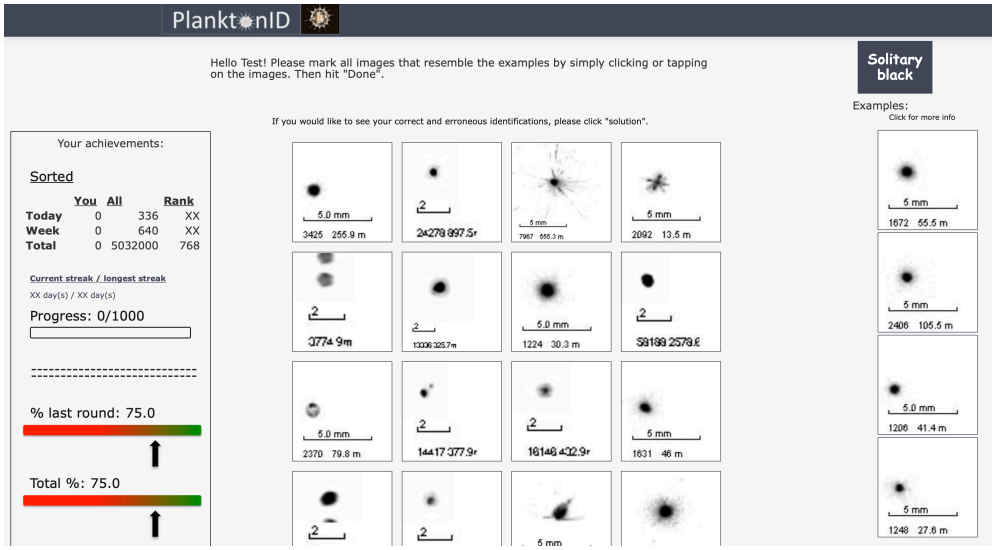

Figure 1.1: The PlanktonID main identification page.

This approach, however, has the following **disadvantages**:

- Because they are outsourcing our biologists' work to untrained strangers, they still need to review that identifications are correct.
- To avoid having experts correct the work of others, the same images are presented to the users so that statistically speaking, the correct identification is performed by the majority of participants.
- Biologists are not completely relieved of their repetitive work.

[1] https://planktonid.geomar.de/en

- When there are incorrect identifications, experts still have to manually correct the species.

- Images must be cropped to show one species at a time, which is a time consuming job, especially taking into account that an expert must do it to correctly crop cases where a species might look like two different ones (and vice-versa).

- There is no apparent management of different projects.

Other platforms like Diatoms [2] or the AMZ [3] work instead as a database of species. While this does not directly help the identification process, it is a useful tool to obtain descriptions and pictures of species, so that experts can help single out difficult images that, for example, are not correctly focused.

As a summary, there are not any platforms that helps to digitize the work scientists do without actually reducing the time involved in this process. Moreover, these applications do not allow third party scientific groups to manage their projects there.

1.3 Objectives

This project aims at providing a web application to centralize, organize and automate everything around phytoplankton identification projects.

While the inspiration of this new platform has in mind aiding the aforementioned biologists, all sorts of projects regarding the identification of species in samplings will also benefit from it.

The platform should provide the following functionality:

- **Taxonomy**: Species are classified in groups according to shared characteristics. These groups can be aggregated to form a containing group, thus creating a hierarchical way of classification of organisms. Every level of this hierarchy is called a taxonomic rank.

  The platform should provide a way to store all the species that will be used through the application, and a sufficient simplified version of their hierarchy.

- **Catalogs**: Any taxonomic rank, regardless of level, can be grouped together into a catalog. Catalogs allow to group those species for projects to use as the available identifiable species that biologists can pick from. In plain terms, they work like *folders* of species or taxonomic ranks that are relevant for a given study.

[2] https://diatoms.org/species
[3] https://www.imas.utas.edu.au/zooplankton/image-key

- **Project management**: The core of the application will be the projects. A project will organize different biologists with a single goal of identification. Withing a project, its creator (also called the Project Manager) will be able to:

  - **Select the catalogs** that will be available for the identifications made in the project to use.
  - **Assign *taggers*** (those that are ultimately in charge of identifing species) to the project. This also includes the possibility of discharging them if necessary.
  - **Upload samples** to be processed. Samples are a set of images taken with a microscope that share a logical unit (e.g. sample of water taken on Doniños Lake on July 4th, 2020).
  - **Create tasks** that will assign the identification of species in the images of a sample. Tasks can assign work to members of the project or automated systems linked to the application. Independently of the nature of the assignees, work will always be equally balanced.

    Tasks will have the possibility of:

    * Create multiple identifications per image, assigning more than one of the assignees to the same image, for double-checking purposes.
    * Have backwards compatibility with previous tasks on the same sample. Backwards compatibility guarantees that the same tagger will not be assigned to an image that has already been assigned in compatible tasks.

- **Authentication**: The work the biologists will carry out must be protected and every change must be traceable and identifiable. A *users authentication system* will be able to always identify who has done a certain action.

- **User management**: Due to the necessity of controlling who has access to the application, administrators will have the ability to create user accounts. These accounts will send an email to the to-be user to proceed with the account creation.

- **Authorization**: Sections and features of the platform must be separated in logical groups accessed only by certain roles, therefore only making it accessible to designated people. These roles will be defined in a hierarchical order, where higher ranks will have all of the permissions of the lower.

  The final roles, in ascending order of authorization rank, are:

  - **Tagger**: A *tagger* is a biologist that can be assigned to projects and complete tasks that are assigned to them by identifying species in images.

- **Project manager**: A *project manager* has the ability to start new projects, upload samples to them, start new tasks and manage their own projects. This role also has all the permissions of a *tagger*.
- **Supervisor**: A *supervisor* has the ability to see and manage all projects, even those that they are not *managers* of. *Supervisors* will also be able to modify the global taxonomy of the application and create new catalogs to be used in new projects. This role also has all the permissions of *project managers* and *taggers*.
- **Administrators**: They have the ability of creating new users. This role also has all the permissions of *supervisors*, *project managers* and *taggers* alike.

Chapter 2

Methodology

There are many development methodologies that could be potentially compatible with this specific project. In the following lines it will be disclosed the reasoning behind the usage of the Scrum methodology for the development stage.

2.1 Why Scrum?

Since the beginning, the application's requirements were not strictly defined. There only was a general idea of the main functionality and the problems that it would solve.

Because of this uncertainty it felt like a right decision to look for a methodology that was of an iterative nature. With small iterations, the definition of the requirements would be able to evolve alongside the development process, with a more precise refinement of what features were needed.

Scrum was ultimately chosen because how well it fit the project. First, because as an iterative methodology, would satisfy the aforementioned needs. But, secondly, because an incremental methodology would also allow to constantly produce a usable product at the end of every iteration, which was ideal due to the also undefined scope of the project and what high-level features would end up being incorporated.

2.2 Scrum

By definition, Scrum is an agile, iterative, incremental methodology where three key factors are involved: roles, artifacts and workflow. Because the development team of this project is formed by only one person and it's motive is a thesis project, it makes sense that the methodology would suffer some deviations from the norm to be adapted to this specific situation.

2.2.1 Roles

Scrum defines a series of roles that members of the project have to stick to:

- **Product Owner**: It is usually one individual that represents the necessities of the customer. This individual is in charge of maintaining the **Product Backlog** via customer-centric terms (i.e. definition in common language).

  This role is held by one of the directors of the project, that will be representing the interest of those that requested the application for their research.

- **Scrum Master**: The role's purpose is to make sure that the philosophy of the methodology is followed by everyone involved. For example, they help the Product Owner to manage the *Product Backlog*.

  This role would be held by the author.

- **Scrum Team**: A team composed from 3 to 9 members that are in charge of the development of the product. They could be developers, designers, testers and any other individual necessary but they act as a team without any defined internal subdivisions.

  Here the deviation is where will show the most. As this project is part of a dissertation, there will only be one developer, the author.

2.2.2 Artifacts

The set of supporting elements that are part of the methodology's process are called artifacts. These are used to organize and manage all development process and to coordinate all members in the team.

Product Backlog

The *Product Backlog* (PB) contains a list of all features to be implemented. Features are ordered by their importance for the product. The format of the PB may vary from project to project. We will be using **User Stories** to describe every feature.

A *User Story* is a brief description of a feature in natural language. It follows the following template:

> As {role} we will be able to {capability}, because {necessity}.

The *Product Backlog* can be (and will be throughout this project) modified to adapt to the evolving necessities of the product. Removing, adding, and even modifying *User Stories* is part of the agile development nature.

Every User Story will be accompanied of the following metadata:

- **Priority**: Some features are more important than others when trying to build a product that is as functional as possible to meet the market's necessities.

  It can, however, also be used as a way to define which features are dependent of the existence of another (it might not make sense to implement an "Edit user login info" page if users are not yet implemented). This is the approach followed for this project.

- **Points**: Points define the amount of work a given feature is worth. Some may be more costly to develop in terms of time or necessary people involved.

  There are many approaches to compute points. We could have used *Function Points* with a well-defined point calculation methodology, but it would be overkill for this specific project's needs.

  Instead we opted for *Story Points*, which are based on pure heuristic opinions of the development team to estimate how much time a story is worth. Because the *Product Backlog* is modifiable, we will be constantly calibrating this parameter.

- **State**: Features can have multiple states: "available", "in progress" and "finished". This is very self-explanatory. When a *User Story* is currently being worked on, it changes to the "in progress" state. Once the development is finished, it finally changes to the homonymous state.

Sprint Backlog

For every iteration (also called sprint in Scrum's terminology), a Sprint Backlog is defined. A *Sprint Backlog* is a small subset of the PB that represents the *User Stories* to be implemented in the current iteration. We used GitLab's repository issue tracking to keep *track* of all of the *Sprint Backlogs*.

2.2.3 Workflow

Sprint

A sprint is an iteration of the methodology that spans for a predefined period of time. In every iteration all classical life cycle stages take place, even testing. With this, we ensure that at the end of every iteration, a potentially releasable product is obtained.

Sprints were defined as **weekly iterations** of the **equivalent work of 35h per week** (approximately eight work hours per day). Every sprint would be given a total of **15 Story Points** to implement. This means that in every *Sprint Planning*, the total amount of points among the chosen *User Stories* for the Sprint Backlog could not surpass this number.

Needless to say, given the context of this project, it could not be possible to stick to new iterations occurring every week. The actual time span of the sprints were artificially increased

to only take into consideration the expected hours of work. Thus, creating another deviation from Scrum's guidelines. This means that if we actually constantly worked 2h per day (being one developer), the actual span of the sprint would be a little more than four weeks.

Sprint Planning

Every *sprint* starts with a meeting of the team. There it would be discussed which elements of the PB would be considered for the sprint and the overall objectives and expectations for it. This event is also used to organize how work would be achieved. In this meeting an objective is set, a **Sprint Goal**, which is the final expectation as the output of the whole sprint.

Daily Scrum

Scrums defines a daily meeting that involves mainly the development team, where every member would share what they did the previous day and what they intend to do the current day. This meeting will not be held because the development team is formed by only the author.

Sprint Review and Sprint Retrospective

At the end of every iteration Scrum expects two meetings to take place. The *Sprint Review* intends to inspect the completed work and present the new functionality to the client. The *Sprint Retrospective* would reflect on the past sprint, providing feedback of the process and trying to follow the continuous improvement philosophy.

In our specific case, these two meetings took place at the same time because of the availability of the participants. For this same reason, every *Sprint Planning* was joined together to these meetings right after.

In these meetings, the development team would give a small presentation of the work done throughout the sprint, all technical obstacles that were found and how they were overcome and, finally, a small demonstration of the features implemented. A tiny period of time would be booked at the end of every meeting to debate about the effectiveness of the sprint and the correctness of the heuristic metrics for *Story Points* computation.

2.3 Configuration Management

To keep track of all the changes and control all versions of the application, especially when it comes to the potential final product that comes as the output of every *Sprint* we will be using Git.

2.3.1 GitLab

Among all the hosting options, we decided to use the **official FIC's GitLab** hosting. Firstly, because it makes sense to use a hosting already dedicated to educational purposes within the FIC, but also because it comes with useful tools that fit agile development pretty well.

In order to keep track of how much time is being spent in one given task (User Story), we will be using GitLab's own issue tracking, which provides tools to push the time spent with every commit as well as the issue the commit belongs to.

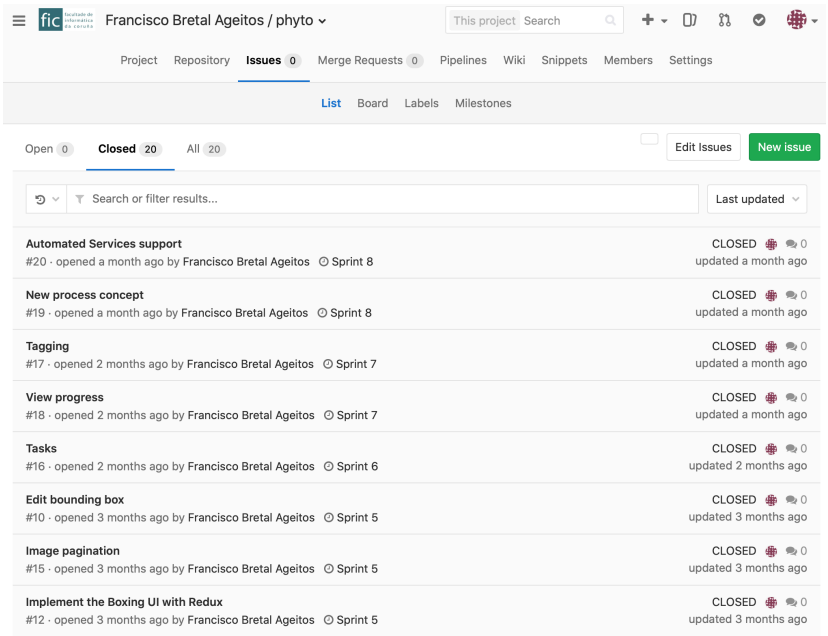

Figure 2.1: The issue list in the repository.

It also comes with Milestone support, which is a great way to organize issues for the different sprints. Moreover, issues have two states (opened and closed), mimicking all the states a User Story can have in the Sprint Backlog. Because of all of this commodities, **issues were used as the Sprint Backlog**.

2.3.2 GitFlow

The workflow that was used to organize our repository was GitFlow. GitFlow defines a strict branching model where the following types of branches are used:

- **Main**: The `main` (previously known as `master`) branch that comes with every git repository. This branch is used to merge into it only final changes that are meant to be live or in production. This is, of course, not used in this project because it never was deployed.

- **Hotfixes**: `Hotfixes` branches are meant to be used to apply quick fixes directly into the `main` branch when serious bug are found. Because the `main` branch is not used, hotfixes where inevitable not used either.
- **Releases**: *Release* branches contain all final versions of a product that are meant to proceed to production. This branch type is used to contain the result of every sprint. It is also worth noting that not every release is necessarily meant to be deployed. It can be also used as a milestone.
- **Feature**: `Feature` branches are meant to contain groups of commits that represent a given feature (for example, a given User Story). When they are finished, they are directly merged into develop with no fast-forward.
- **Develop**: The `develop` branch (there's only one) will contain all of the features of the application. This is also the branch where `release` branches are merged from (and even into).

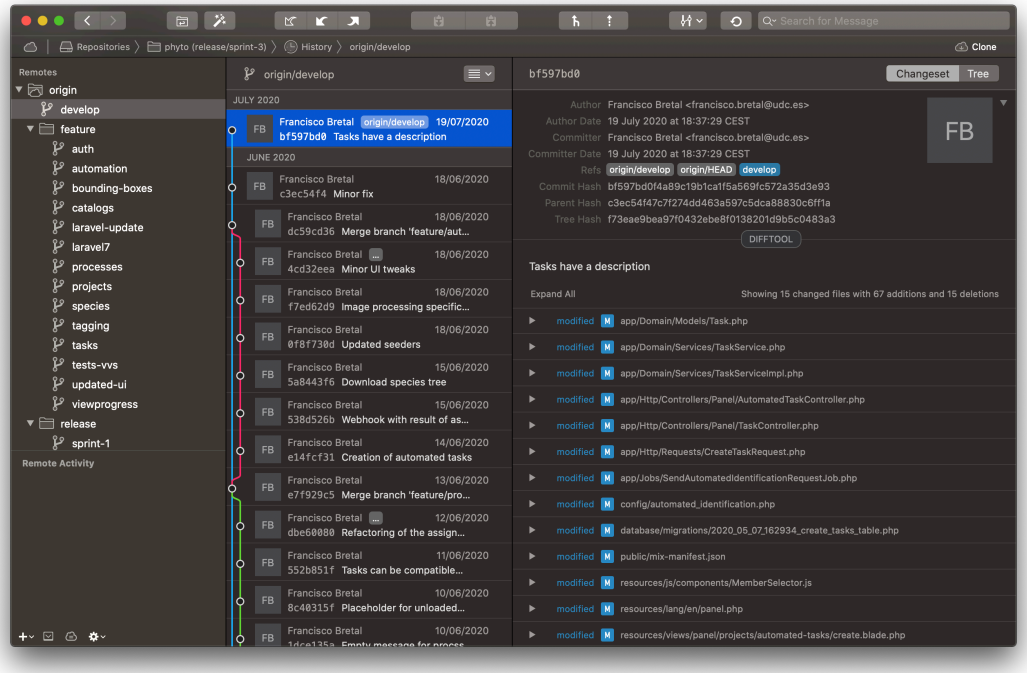

Figure 2.2: A quick look at the development branch as of the end of the project, with some of the feature and release branches on the left.

Chapter 3
Technologies

Choosing a technological stack that will accompany us for the duration of an entire project is not something that can be taken lightly. It is very important to dedicate a few moments to analyze the scope of the project, its requirements and what different benefits technologies could provide.

3.1 Backend

3.1.1 PHP

PHP (PHP: Hypertext Preprocessor) is an **interpreted scripting language** that is especially suited for web development [2]. Since its first release in 1995, it has evolved from a dynamic template engine to a fully-fledged programming language with features such as **gradual typing** [3], improved single inheritance with traits and excellent performance, among other great capabilities.

PHP was chosen as the main backend language because it fits well with the agile methodology, thanks to the flexibility it provides and its interpreted nature. Additionally, the development team had already a lot of experience with the language.

3.1.2 Laravel

Laravel is a very popular web framework in the PHP community. The framework follows the common **MVC** design pattern, provides features that follows PHP's PSR recommendations and promotes the use of other software patterns like the **facade pattern**, which is its signature.

Like Java's *Spring Framework*, Laravel prioritizes **conventions over configuration**. Because of this, Laravel can reduce the amount of code a developer would need to do simple tasks, and, hence, the reason why it was chosen. Again, the development team was already

familiar with the framework, which helped with decision-making.

This project will be using Laravel 6 as it's the most recent LTS version available.

Eloquent ORM

Laravel's out-of-the-box ORM is Eloquent. Eloquent is an ORM that follows the **Active Record (AR)** pattern [4]. The AR pattern wraps database tables in classes that can interact with the data in them [5].

Blade

Blade is the default template engine that comes with Laravel. Because PHP was initially created as a template engine, **Blade brings syntactical sugar** that is compiled into native PHP code and then cached, thus avoiding any possible overhead that a template engine could bring [6].

3.1.3 Node.js

Node was also used to develop simple dummy services used as example for the automation features of the platform. It was chosen for its particular swiftness when developing Rest APIs.

3.2 DBMS

Being the backend built with Laravel's default ORM Eloquent, there is actually support for all DBMS that are supported by Eloquent:

- MySQL 5.6+
- PostgreSQL 9.4+
- SQLite 3.8.8+
- SQL Server 2017+

Because the application did not have the need to create custom non-standard SQL queries and were completely created under the ORM's abstraction, support for all previous DBMSs still theoretically holds. However, MySQL was the DBMS that was aimed at providing support for the application and the only one tested.

It is worth noting that **SQLite was used during development**. In contrast to MySQL, SQLite is not a client-server database engine, it is directly supported by PHP and holds all the data in a single file. This makes it very easy to setup and to work with, perfect for development environments.

3.3 Frontend

3.3.1 SASS

SASS is a **preprocessor scripting language that compiles into CSS**. SASS provides two different syntaxes to create CSS code: SASS and SCSS. The latter was the syntax used for this project, being the newest and also the most standard because of its nested metalanguage nature (any CSS code is already valid SCSS code).

SCSS is used to avoid code repetition and create an overall more **maintainable and legible CSS code**, providing features like variable support, static file importing, procedural scripting, color mutation features, style templates and many more.

3.3.2 Bulma

The CSS framework that will be used is Bulma. Bulma is very similar to Bootstrap, but it is lighter and more modern by being built on top of the new CSS Grid and Flexbox standards. Moreover, it doesn't require jQuery or any JS code at all, which remove the load of unnecessary scripts in the frontend and makes the framework more lightweight. Altogether, it is a simple an useful tool to build styles on top of it.

3.3.3 React

React is a very popular frontend JavaScript library that is used to build dynamic user interfaces. Even though it is mainly used to create SPA websites, its paradigm is just centered around rendering data to the DOM. This allows the use of React in combination with other libraries or technologies in *classic-style* websites.

The latter approach was chosen for this project. Given that we are using Blade as a template engine we will only need React to create complex user interfaces only found in certain sections.

Overall, we are using React because it improves the code quality by creating declarative, easy to read, reusable components.

3.3.4 Redux

Some of the most complex interfaces require of in-page state management to keep track of how the user can interact with the UI. We will be using Redux, a tool specifically created for this.

Redux is a library that is very common to be used hand in hand in React applications. It brings predictable containers using a reducing-function style to keep the state inmutable, which ultimately helps to create performant, bug-free code.

3.4 Validation and Verification

3.4.1 Unit Testing

To test the application PHPUnit was used. PHPUnit is the most used unit testing framework in the PHP world. It comes with all the features you would expect, with support for setting up testings and creating testing environments.

3.4.2 Mocking

In order to mock class behavior when testing we will be using two different technologies:

- **Facade Mocking**: By default, Laravel provides facades that are used as interfaces for different APIs. These facades are easily mockable out of the box thanks to Laravel Facade Mocking.

- **Mockery**: For every other custom made class we will be using Mockery, a PHP mocking framework.

3.4.3 Random Data Generation

When testing we will be using Faker to generate fake data that will be useful when creating variadic tests that will greatly improve the quality of the modules to be tested.

This library is also used to populate the database with dummy entities that can help with the overall understanding of the application when in the development stage.

Profiling

To profile the application and be able to monitor the amount of memory is being used, the amount of SQL queries are being fired and so on, we will use PHP Debug Bar.

Mutation Testing

In order to ensure that the available tests can detect the most common mistakes we will be using Infection, a mutation testing framework that can edit the application code to check whether the current tests can detect those changes and fail as consequence.

3.5 Development Tools

3.5.1 Version Control System

All changes to the project throughout its development have been stored under a GIT repository. It is hosted in UDC FIC's GitLab. By using GitLab we also get some other features like *Issue Tracking*.

3.5.2 Integrated Development Environment

Jetbrain's PhpStorm was used as the IDE to develop this project, which comes with features like autocompletion, static analysis of the code, refactoring and many great features to improve the development experience.

3.5.3 Debugger

By default PHP does not have a debugger where you can create breakpoints, monitor the call stack and conditionally stop execution. It relies in old-style yet improved print-style functions to do all of this.

While in most cases this would be sufficient, we decided to install a **fully fledged debugger** called XDebugger, as an extension to the PHP interpreter. This and some extra configuration allowed to link PhpStorm to the interpreter and be able to debug as expected.

3.5.4 Composer

Composer is a package manager for PHP that allows to easily import third party libraries and *autoload* them to the project. PHP needs to explicitly include all files that are needed in the project (not to be confused with Java's `import` for classnames, which in PHP is called `use`), so *autoloading* does that for you, automatically.

3.5.5 Laravel Mix

Laravel Mix is a wrapper around Webpack, a bundler that will be used to **bundle, minify and transpile frontend assets**. Laravel Mix provides a simpler interface to do most common things webpack is used for.

This technology will be in charge of taking care of React's ES6-style JavaScript code and transpiling it to its ES5 version, vastly supported by most modern browsers.

It is also used to bootstrap SASS into the bundling process, so the SCSS code will be automatically transpiled into CSS code and then minified to be production-ready.

3.5.6 Laravel Valet

Laravel Valet is a development environment **based on the nginx server**. It automatically serves projects under a custom TLD for local development. This is all done without the need to manually modify the hosts file of your computer [7].

3.5.7 Mailtrap

Unlike any of the technologies mentioned above, Mailtrap is an online service. Mailtrap is an SMTP testing server that *traps* all outgoing mails generated by the platform and collects them into a single mailbox. This is very useful for testing mailing without having to actually send emails to real addresses.

Chapter 4
Planning

Choosing agile development was clear because of the unknown details of the project and the high variability that it could potentially have. As consequence, it was not possible to have a very detailed planning since day one. Every iteration can modify the requirements to adapt to any future necessity that was not taken into account, making estimations very hard.

In the following lines we will be discussing the initial planning, with the design of the Product Backlog, as well as how it evolved throughout the sprints.

4.1 Product Backlog Design

The very first step in the development process was to design a *Product Backlog*. Designing a Product Backlog acts as a first global step to analyze the domain and create, at least, an overall idea of the necessary requirements and the final product that is expected to produce.

An initial Product Backlog, would help to understand the overall objectives and ideas the team had for the resulting application. Of course, requirements were not strictly defined, so it was expected for the PB to change throughout development.

4.1.1 Roles

However, before defining any *User Story* it was necessary to **analyze the domain and identify the different roles** that would take part in the user pool that would make use of the application.

We found that the best approach would be to use a hierarchical set of roles, where a high-order role would have all of the permissions the roles beneath would have.

Next up, a list of the final roles is provided in ascending importance order:

- **Tagger**: Taggers are the core workers of an identification project. These are the biologists that work on water samples to find and identify phytoplankton species.

- **Project Manager**: They are those people that are in charge of managing specific projects of identification. For example, a project that was created to analyze the waters of all the lakes in the province of A Coruña.
- **Supervisors**: Supervisors were imagined as a special role just for the application. They are individuals who have the ability to change the species database, as well as the groups of species (catalogs) that project managers can use for their projects.
- **Administrators**: This is the most capable role in the application. It has permission to invite new users to the application and assign a role to them.

4.1.2 Product Backlog

With all the roles involved in the application already defined, it was now time to create *User Stories*. For this, a meeting was held with the Product Manager and the help of the Scrum Master.

Table 4.1: The initial Product Backlog.

ID	User Story	Points	Priority
01	As an administrator we must have **access to an Administration Control Panel** so that they can manage all aspects of the application.	3	1
02	As an administrator we must be able to **manage user accounts** to allow access to the application to other people.	1	1
03	As an administrator we must be able to **assign any role to a user or take them away** so that they can control who has access to the multiple parts of the application. (US-3)	3	1
04	As a supervisor we must be able to **manage catalogs** so they can create the containers in which the different animal species can be organized in and assign them to a specific project.	5	2

Continued on next page

Table 4.1 – *Continued from previous page*

ID	User Story	Points	Priority
05	As a supervisor we must be able to **manage species that are hierarchically organized** so we can add, remove, list or update the different species to be assigned to a catalog to further identify them in the pictures.	8	1
06	As a project manager we must be able to **create new projects and manage or delete our own** in order to be able to upload pictures from different samples that are bound to a specific purpose.	5	3
07	As a project manager we must be able to **assign users to our own projects** so that they can access the samples and its resources and help with the identification process.	5	3
08	As a project manager we must be able to **manage samples within a project to then be able to upload pictures and assign them to the samples**.	13	3
09	As a project manager we shall be able to **manage tasks with specific types**, which will be assigned to one or more users and that represent a job within the project to be completed by those users.	5	4
10	As a project manager we shall me able to **assign a task of type "creation of bounding boxes" to users and to a set of pictures**. The task should equally divide the work between the assigned users. The project manager must also be able to provide the number of times a picture should be reviewed for bounding box identification, so that the task can assign the same photo more than once to different users. Once a "creation of bounding boxes" task is finished, it results in multiple sets of bounding boxes for each picture, one for each user that has created bounding boxes in the picture.	21	4

Continued on next page

Table 4.1 – *Continued from previous page*

ID	User Story	Points	Priority
11	As a project manager we should be able to **see the progress of completion of a "bounding box creation" task**. This progress will be computed according to how many pictures has been set as "done" by the *taggers*.	3	4
12	As a project manager we must be able to **assign different sets of bounding boxes (previously created on a "creation of bounding boxes" task) of a project to a given "species identification" task**, so this task will be able to track the progress of completion of the identification of the bounding boxes in that sample.	3	5
13	As a project manager we must be able to **assign different sets of bounding boxes (previously created on a "creation of bounding boxes" task) of a project to a given "species identification" task**, so this task will be able to track the progress of completion of the identification of the bounding boxes in that sample.	3	5
14	As a project manager we will be able to **assign one or more catalogs to our project** so the *taggers* will have a list of species to choose from only from those assigned catalogs.	3	4
15	As a tagger we will be **assigned to project tasks by projects managers** in order to create bounding boxes or to tag them to a species.	1	5
16	As a tagger with permissions to create bounding boxes in a specific project we will be **able to create bounding boxes on an image from a task that is currently in progress and that we are assigned to** so that, in the future, others can identify the species of the animal in that bounding box.	13	5

Continued on next page

Table 4.1 – *Continued from previous page*

ID	User Story	Points	Priority
17	As a tagger with permissions to identify species in a project will be able to **tag a bounding box with a species from the catalogs assigned to the project** only if we were assigned to a task of the identification tag and it's currently active.	7	5
18	As a tagger that has been assigned to a "creation of bounding boxes" task we must be able to **set a picture status to "done"** to mark our work as finished in that picture.	3	6

As said in section 2.2.2, the strategy used to compute Story Points is completely heuristic. The development team (in this case, just the author) estimates the amount of time a given task is worth according to their previous knowledge of the technologies being used and the task at hand. Then, an equivalent point number is obtained, proportional to the points-hours relation explained in section 2.2.3.

4.1.3 Non-functional requirements

Alongside with the Product Backlog, there were some **non-functional requirements** defined that would not belong as such within the PB but should be taken into account throughout development.

While there is many approaches regarding the inclusion of non-functional requirements within the Scrum framework, we decided to opt with the simplest solution because our requirements were very generic and very few: We took the non-functional requirements into account when implementing every single User Story and we never removed them from their special place in the Product Backlog.

The following list contains all of the non-functional requirements for the application:

- The application must be **scalable**: adding new features should be easy to do, without having to restructure the code.
- The application should be **easy to use**.
- It should follow **common UI/UX design patterns**.
- The application must be localized, supporting **internationalization**.

Resource	Units	Individuals	Cost	Total
IDE License	1	-	199€	199€
Software Engineer	245 mh	1	20€/h	4,900€
Meetings	7	3	30€/sprint	630€
Computer	1	-	800€	800€
				6,529€

Table 4.2: The base cost estimation for the project.

4.2 Estimation

With the initial Product Backlog in place, we can estimate how much time the project is going to take, as well as how much money will it cost. This is, nevertheless, only valid as an initial estimation, as deviations are expected to happen.

Having a total amount of story points of **105**, and sprints of **35h** and **15pt**, we can compute the total amount of hours:

$$Time = 105\ points \times \frac{35\ hours}{15\ points} = 245\ hours$$

Similarly, we can compute the expected sprints:

$$Sprints = \frac{105\ points}{15\ points} = 7$$

As seen in table 4.2, to compute the economic cost of the project, we are going to assume the average cost by the hour of a developer to be 20€/h to account for salary, fees and other possible costs for an hypothetical business. Likewise, the price of the computer is assumed to be the average computer price.

Of course, we are not taking into account server costs of the GitLab hosting or any possible deployment server of the application.

In order to compute the cost of every meeting we must take into consideration the following:

However, we *will* be considering the cost of every meeting that was held, giving them a time estimation of 30 minutes each. As they all took place one after the other, it would be a total of 1h 30min for every sprint (Planning, Review and Retrospective). Every participant is payed 20€h and, because every meeting lasts 1.5h, the total cost by sprint is of 30€/sprint.

4.3 Sprints

After designing the product backlog, the first Sprint planning could be held to start the development process. We will be going along all the sprints that took place and what effects they had in the overall development of the project. Of course, because Scrum is an iterative methodology, all of the classical software development life cycle steps are repeated in a smaller scale for every *Sprint*.

This means that for every sprint, a User Story would be **analyzed** to understand the necessity it tries to fulfill to correctly **design** the right feature, before actually **implementing** it and finally **testing** it. The usual steps we all know and love.

4.3.1 Sprint 1

The first *Sprint Planning* meeting defined the foundations for the application. It was crucial to select those User Stories that would define the frameworks and tools from which the rest of the functionality would be built upon.

Table 4.3: Sprint 1 Backlog.

Ref.	User Story	Points
01	As an administrator we must have **access to an Administration Control Panel** so that they can manage all aspects of the application.	3
02	As an administrator we must be able to **manage user accounts** to allow access to the application to other people.	1
03	As an administrator we must be able to **assign any role to a user or take them away** so that they can control who has access to the multiple parts of the application.	3
05	As a supervisor we must be able to **manage species that are hierarchically organized** so we can add, remove, list or update the different species to be assigned to a catalog to further identify them in the pictures.	8
	Total:	**15**

Throughout the span of this sprint the Administration Panel was created from scratch. The

administration panel was imagined as a place where all high-order settings and management options would be placed. Simply put and despite its unfortunate naming, it is the place where all roles except the *taggers* would interact with the application to set everything up for the *taggers* to do their job.

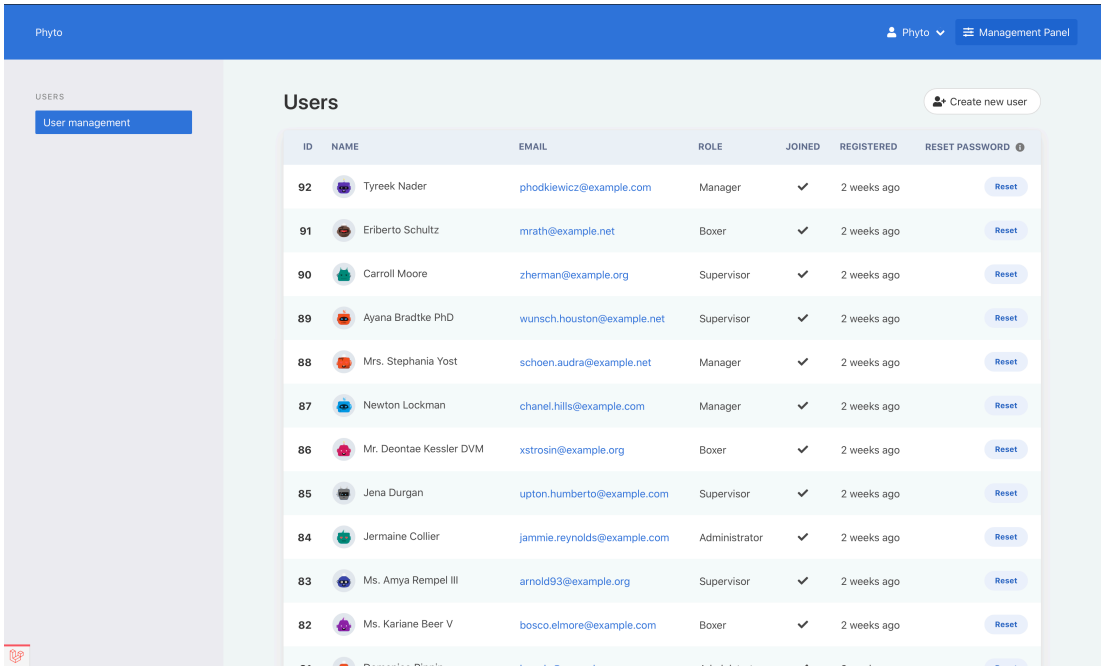

Figure 4.1: The control panel as of Sprint 1.

This sprint's objective could be summarized in three main goals:

- Create the administration panel framework that provides scalable tools to easily build more modules.
- Create the very first module, the user management module that would allow administrators to create accounts.
- Create another module to manage the application's species with an interface (called Taxonomy UI) that would show the taxonomy tree and would allow us to interact with it by adding nodes and editing the existing ones.

Some of this sprint's challenges were the implementation of authentication and authorization, because of their technological consequences regarding email sending. We will be getting into detail on this in sections 5.2.1, 5.2.2, 5.2.3, and 5.2.4.

Sprint Review and Sprint Retrospective

We introduced React into the technological stack of the application in this sprint because it was a great tool to build the Taxonomy UI. This, however, was not expected and heavily increased the time spent in the implementation of the corresponding user story. Due to the lack of knowledge of the technology by the development team, **task story points were greatly underestimated**.

After both meetings, it was concluded that the estimation for some of the tasks were a little off. As consequence, **the sprint lasted more than it should**. Usually, the sprint would be forcefully ended, and the unfinished job would be thrown back to the Product Backlog as new *User Storys* adjusted to the work left. However, the extra hours were not worth to end the sprint and it was artificially increased in size as an exceptional matter.

Because of this, it was necessary to tweak some of the *User Stories*'s estimated points because they were both overestimated and underestimated in some cases. For example, the Catalog management *User Story* was lowered from five points to just three because it would be basically a simple entity with CRUD-like behavior.

4.3.2 Sprint 2

The Sprint started with the following Sprint Backlog. Some of the stories were new in the Product Backlog as a result of the Sprint Planning meeting because some of the features needed more defined behavior to satisfy the Product Owner's needs.

Table 4.4: Sprint 2 Backlog.

Ref.	User Story	Points
04*	As a supervisor we must be able to **manage catalogs** so they can create the containers in which the different animal species can be organized in and assign them to a specific project.	5 ⟶ 3
New	As a supervisor we must be able to **create a new catalog from an existent one**.	1
New	As a supervisor we must be able to **seal** a catalog to make it not modifiable and also available for use in projects. This would also allow catalogs to be in edited mode and also in obsolete mode to disable them forever.	1

Continued on next page

Table 4.4 – *Continued from previous page*

Ref.	User Story	Points
06	As a project manager we must be able to **create new projects and manage or delete our own** in order to be able to upload pictures from different samples that are bound to a specific purpose.	5
07	As a project manager we must be able to **assign users to our own projects** so that they can access the samples and its resources and help with the identification process.	5
	Total:	**15**

The main objective can be summarized in two main goals:

- Implement the **catalog management** module in the administration panel that would allow to create catalogs, containers of groups of species that can be used in projects.
- The creation of projects with the basic metadata that would act as the foundation for the next User Stories in coming sprints.

This sprint went smoothly because all the features to implement were very CRUD-like. No part of the development really stands out and the calibration to the estimations done before the start of the sprint looked like it helped to correct the deviations of the estimations.

4.3.3 Sprint 3

This sprint would implement one of the cores of the platform: file upload. For this, it was necessary to dedicate the sprint mostly to it to do it right. Consequently, the Sprint Backlog was small.

We decided to add a new Technical Story to the Product Backlog because at the time of these events, a new **LTS version** of the framework came out with some useful features for development. For this, we decided to upgrade it to the newer version. Laravel updates are quick and simple, specially if you do not extend most of the default behavior, that is why it is worth only one story point.

Throughout this sprint we implemented the creation of samples form, that allowed the upload of one or more sample images. It is worth remembering that a water sample can have multiple images that represent different pictures taken with a microscope.

Table 4.5: Sprint 3 Backlog.

Ref.	User Story	Points
08	As a project manager we must be able to **manage samples within a project to then be able to upload pictures and assign them to the samples**.	13
New	Upgrade the framework the latest version.	1
	Total:	**14**

We will be covering the main challenge of this sprint, the upload system and the image management in section 5.2.5.

4.3.4 Sprint 4

Sprint 4 can be considered a milestone in the project's development. Its goal is to build the foundation of the functionality that allows the creation of bounding boxes.

In the Sprint Planning meeting, it was necessary to do a rework on some of the stories, specifically, those stories around project "tasks". The story regarding tasks for bounding boxes had higher estimation points than what a sprint is worth. It was necessary to **divide it into multiple stories**. As a result of the division, a new user story of 15 points would be the only story in this sprint's backlog, and everything else regarding "tasks" was left for later in the Product Backlog.

Table 4.6: Sprint 4 Backlog.

Ref.	User Story	Points
New (10)	As a tagger we must be able to **create bounding boxes** in images so we can use them to identify species in samples.	15
	Total:	**15**

Bounding Boxes are rectangles drawn on top of images that are used to surround areas of interest. In our case, these areas would be the specimens found in a sample image.

Creating these bounding boxes is the main objective of the application and it is what *taggers* are going to do most of the times. Everything involved with its development will be covered in section 5.2.6.

Sprint Review and Sprint Retrospective

When this sprint came to an end, we could not finish the only user story assigned to it. Even though that User Story was already rethought in the beginning of the sprint to divide it into multiple stories, we still did not quite estimate the actual work behind it.

Parts of the creation of bounding boxes like editing them and removing them were supposed to be included in this story, alongside some minor quality of life improvements. These had to be added back to the product backlog to be continued in the next sprint.

We took advantage of the development of the Boxer to get a refined understanding about

4.3.5 Sprint 5

After the forceful ending of the previous sprint without completing everything that was expected to do from the Sprint Backlog, we had to continue with it in a new sprint. The work left was made up of new User Stories that were added to the PB.

Table 4.7: Sprint 5 Backlog.

Ref.	User Story	Points
New	As a tagger we must be able to **switch between different images with a carousel** that eventually will show whether an image has been tagged or not.	1
New	As a tagger, when we select a bounding box in the UI, it should be shown in the sidebar which one it is, scrolling if necessary.	3
New	As a tagger we must be able to edit an already created bounding box.	7
New	As a tagger we must be able to **delete an already created bounding box**.	1
New	As a tagger, we must be able to see in the sidebar a **preview of the bounding box** for each one of them.	3
	Total:	**15**

Basically, the main objectives of this sprint were:

- Implementing an **edit** and **remove** option for bounding boxes.
- Create a **pagination with thumbnails** (the carousel) to navigate through different images.
- Clicking in a bounding box in the canvas should **highlight it in the sidebar**.

The challenges of this sprint are shared with the previous one in section 5.2.5.

4.3.6 Sprint 6

After the previous sprint successfully closing the implementation around the Boxer, it was time to jump into something different. In the beginning of Sprint 4, we divided the stories

related to the creation of bounding box tasks, to the support for the creation of bounding boxes and then the support for the creation of tasks. This sprint would be focused on implementing the latter.

As always, we took advantage of the Sprint Planning to analyze into more detail the necessities of the Product Owner, extending the available user stories if necessary. As consequence, the concept of different tasks that was initially considered (see user stories 10 and 12) was discarded on favor of just one type of tasks.

Table 4.8: Sprint 6 Backlog.

Ref.	**User Story**	**Points**
New (10, 12)	As a project manager we should be able to create tasks to assign users to work on a sample. Each task should start an identification process to the sample, which can be repeated any time. Assignments for every image of a sample will be equally distributed among assignees, balancing the workload.	8
New	As a project manager we should be able to set the number of times an image must be worked on by different assignees, to avoid any possible errors.	1
New	As a project manager we should be able to make a task compatible with previous tasks, which will grant the ability to avoid a same assignee to work on an image that they've worked before.	4
	Total:	**13**

The amount of story points for this sprint did not meet the sprint points cap, but there was not any other story left with the necessary points to fill the gap. On the positive side, the extra time can be seen as safeguard for any possible unexpected delays.

The purpose of this sprint was to be able to create a way for *project managers* to make the *taggers* work on the samples they have uploaded. Summarized:

- A project manager would create a task to assign a sample to one or more members of the project.
- If necessary, the project manager could make more than one member to work on the same image, to avoid errors and have multiple opinions of experts.

- Every time a task is created for a sample, an identification process is started, a logical container where the work can be seen in real time by the project manager.
- The application should do on its own all of the assignments, following these rules:
    1. An assignee cannot work on the same image unless a new task is started and is not set as compatible with another where the user has already worked on that image.
    2. Workload should be equally divided by all the assignees unless it interferes with the previous rule.

We will be getting into detail on how we implemented workload balancing in section 5.2.7.

4.3.7 Sprint 7

Sprint 7 was the last sprint dedicated to the core functionalities. The end of this sprint would provide a completely usable platform for biologists to start working on their projects.

Table 4.9: Sprint 7 Backlog.

Ref.	User Story	Points
16*	As a tagger with permissions to identify species in a project will be able to **tag a bounding box with a species from the catalogs assigned to the project** only if we were assigned to a task of the identification tag and it's currently active.	13 ⟶ 7
18	As a tagger that has been assigned to a "creation of bounding boxes" task we must be able to **set a picture status to "done"** to mark our work as finished in that picture.	3
11*	As a project manager we should be able to see the progress of any identification process started by a task.	4 ⟶ 3
	Total:	**14**

Now that tasks were implemented and worked correctly, it was necessary to provide project managers with a way to see the progress of the work done by *taggers*. To see the bounding boxes created by a *tagger* we reused the interface of the Boxer. Thanks to React's reusability, we created a version of the component that is not meant to be modifiable.

Moreover, for every process, its progress is shown, according to **how many assignments**

a user marked as complete.

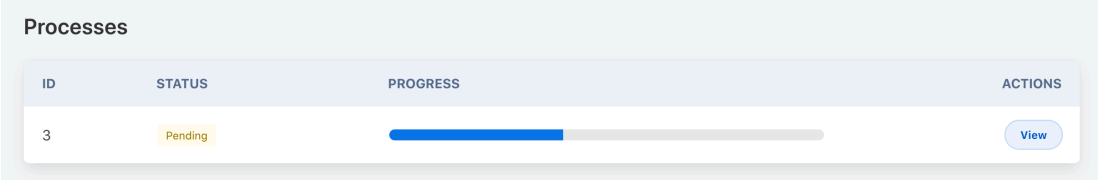

Figure 4.2: The list of processes that a task has started. For every process a list of their assignments can be accessed by clicking on the "View" button.

In addition, we provided a view for every task, to see every process that it has started. Similarly, the sample list from the project management section was added a new page to directly access any process started for that sample.

Finally, this sprint also implemented the ability to select which species a bonding box is identifying. Thanks to the interfaces and the tools we already built, creating this feature was very simple. We used the edit mode of bounding boxes to provide a way to trigger this action. Then, we used the taxonomy UI to create a new version that allows to pick one of the elements.

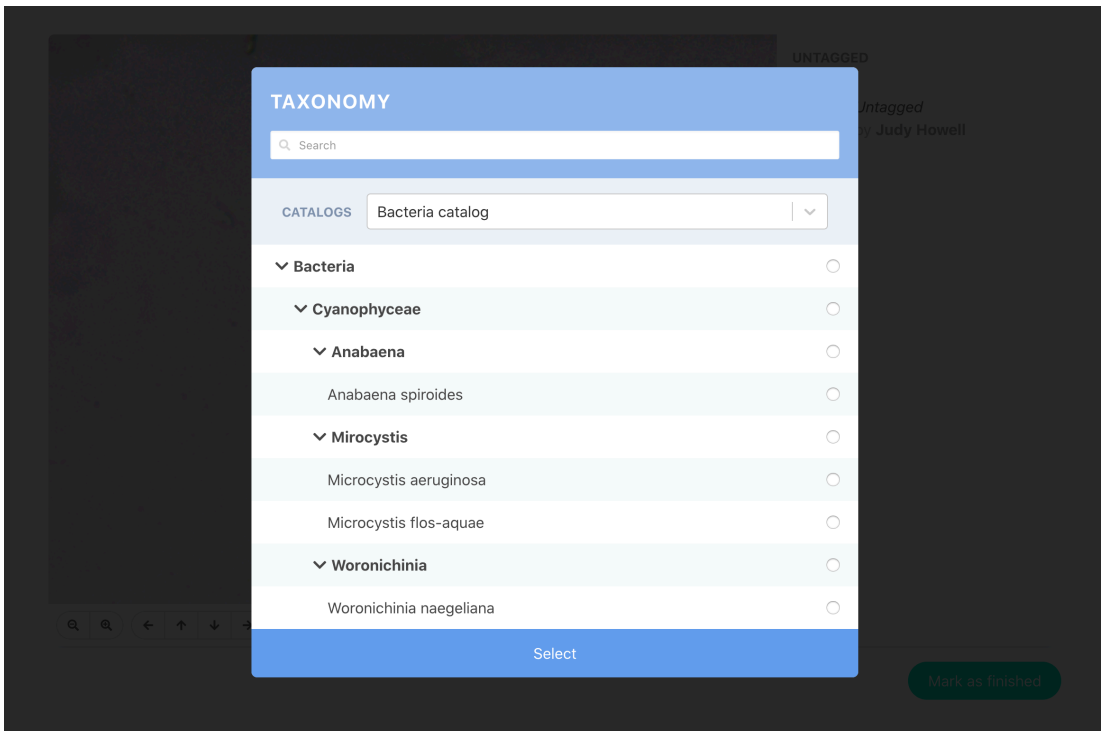

Figure 4.3: The taxonomy picker that is shown to select the identified species. It comes with a selector to change between the project's active catalogs.

4.3.8 Sprint 8

This sprint would be dedicated entirely to implement support for automated tools that would identify on their own species on images.

Due to the available time after the previous sprint, even though all stories from the product backlog were finished, we decided to add a new one to implement automation.

As always, we used the Sprint Planning meeting to imagine how this new feature should be.

Table 4.10: Sprint 8 Backlog.

Ref.	User Story	Points
New	As a project manager we should be able to create automated tasks to bring some automation to the work of the *taggers*. These tasks would assign the identification to automated services.	15
	Total:	**15**

Automated services are tools that are created by other developers[1] that do the **repetitive tasks** taggers usually do: they identify relevant species in images and they *tag* them with the according species.

An automated task would work just like a normal task, but instead of assigning the work to taggers, they assign it to the automated services.

[1] https://ruc.udc.es/dspace/handle/2183/24894

Chapter 5
Development

With the platform already finished, we will discuss some of the design choices and challenges that took place in the development process. The high level architecture and testing techniques will also be covered in this chapter.

5.1 Architecture

The application follows the Laravel MVC architecture, with a custom addition on top of it to add a multilayered architecture, divided in the following:

- The **Domain Models** layer, which contains the representation of the domain, its models.
- The **Business Logic of the domain**, that acts as a service layer that provides the functionality of the application according to the needs of the client.
- The **application layers**, which are basically Laravel's endpoints and serve both the actual application through HTTP via a web application and the shell.

Regarding Laravel, it follows the following directory structure, summarized:

- `app/`: Contains all code regarding the application. Here, subdivisions like `Http/` and `Console/` are found to divide the actual two applications that are found, one accessible from command line and one by HTTP.
- `resources/`: Contains all assets related to the frontend in their pre-compiled state. Here the views are found among the raw JS and SASS (CSS) code.
- `bootstrap/`: This directory contains files that are used to bootstrap the framework when initializing it with a request.

- `database/`: Here the migrations are stored, and also all seeds and factories used to both initialize the database and to create dummy models used in testing and developing alike.
- `config/`: This directory contains all configuration files. These are not **usually** edited directly, because configuration is actually retrieved from environmental variables.
- `.env.example`: A clean version of the `.env` file that must be created to initialize the environmental variables. Here you can tailor your configuration according to your environment (e.g. production).
- `tests/`: All tests of the application.
- `public/`: The entry point to the applications that a web server must serve. Here `index.php` is found, which is the single file that starts all of the request lifecycle. In this folder are also stored the compiled assets from `resources/` and some images or fonts the application might need.
- `storage/`: The storage of the application. Here you can find log files that are outputted by the application, all files the application generates (e.g. sample images) and any other stored file needed for the application itself.
- `vendor/` and `node_modules/`: Autogenerated directories that contains all dependencies that the application and the framework needs to actually work.

5.2 Challenges

The development process was full of design and implementation challenges that the development team had to overcome by making strong design, implementation and technological decisions. In the following sections we will be discussing some of the most notable.

All of the challenges here will be listed in chronological order, from the first sprint to the last one. Taking this into account, it will be possible that some features will not be mentioned or shown because they have not been yet implemented.

5.2.1 Auth

The first sprint was dedicated to build the **foundations for Authorization and Authentication**; after all, for the User Management module of the administration panel to make sense, authentication should be implemented first.

Because this is a very common functionality to have in web applications, there already was support for it as a first-party package. Having authentication and authorization available was

just a few keystrokes away. However, while auth was quickly implemented, it was necessary to do some modifications to the main package:

The default configuration of the authentication system that comes with Laravels allows users to create accounts by themselves, providing a public registration page for everyone.

As it is a requirement for the application to have a private nature, where the members accounts are created manually by the administrators, some of the authentication routes had to be removed and the registration was moved to the administration panel, implementing an invitation system where users would be able to **continue** the user creation process by choosing their own password and activating their account via email.

5.2.2 Protecting Invitations

Among the tweaks done to authentication, it is worth mentioning how user invitations were imagined. Administrators would create accounts by providing the user's data (their name, email, and role) but **it would be the user's job to choose their own password**.

To do this, they would receive an email with a link to activate their account as well as to set their password. This link however, must be taken into consideration carefully: because it is sent as an email (hence, the route is public), and because HTTP is stateless, we do not have any native way to **make sure that the activation page does not suffer a request tampering attack**.

To protect our application against this kind of attacks, we **signed the URL by appending a unique hash in each link** sent in the activation emails, and gave it an expiration date. The ability to sign URLs is supported by Laravel out of the box, so it did not require much work from our part.

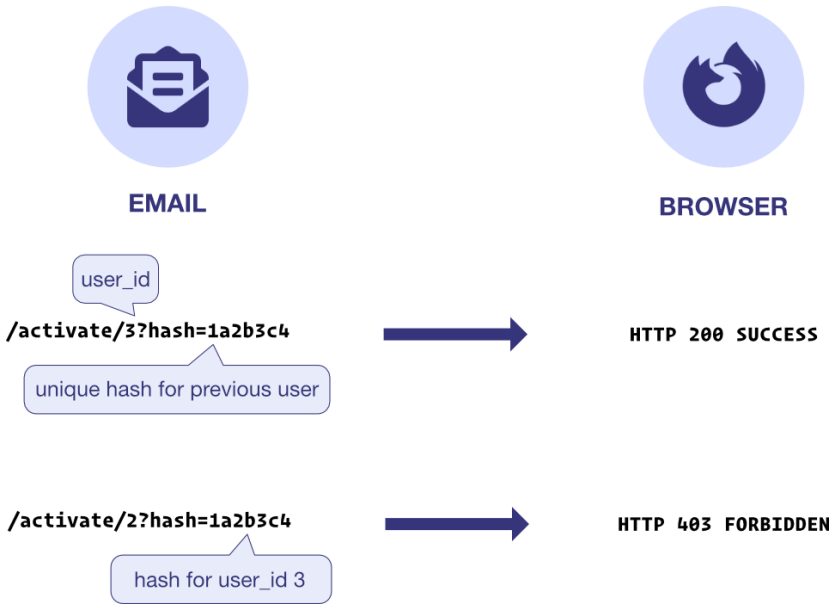

Figure 5.1: Explanation of how signing URLs protects us against request tampering.

5.2.3 Sending email

Sending email had their own problems too. The sending process is a consuming task on its own and, when done synchronously with the request thread, it can hang the application while sending it, greatly reducing the request time.

On top of that, PHP is not multithreaded by nature. Every request is run in their own thread. This would mean that if we wanted to do asynchronous tasks, we would have to find our way around it.

Thankfully, Laravel implements an interface for queues, which is a PHP script being run in parallel that can even have multiple instances if necessary. Asynchronous tasks are then created in, for example, a Redis database or, in our case, for simplicity, the proper application database, and fetched by these scripts that take care of running them.

This greatly improved the response time of the user creation request because the email is not sent by the main thread, but in parallel by the queue.

5.2.4 React

When we were tasked to implement the taxonomy UI in the first sprint, we realized that the interface would be very interactive and it would benefit from AJAX so that the user would not have to load a new page for every change.

Taking a look at the Product Backlog we could see that there would be many other aspects of the final product that would benefit form high interactive UI designs. That's when we decided that it would be a great idea to introduce React into our application's technological stack.

React would help to ensure we followed the non-functional requirement of scalability. We knew that we could reuse the component of the taxonomy UI in other places of the application that we knew about beforehand (because of the Product Backlog). For example, it would be used to select species from the taxonomy when creating a catalog or for a tagger when identifying a species in an image.

It was also a great decision because it is a very dynamic and interactive interface that would benefit from AJAX requests without having to constantly load a new page. Thus, making the creation of new nodes in the hierarchical taxonomy a simple task through the use of popovers, or searching through the tree by simply typing into a filtering text input.

5.2.5 Upload System

Implementing a good uploading system was no easy task. This feature would be used constantly by the Project Managers to upload very large files with lots of pictures. Such a core functionality must be carefully designed.

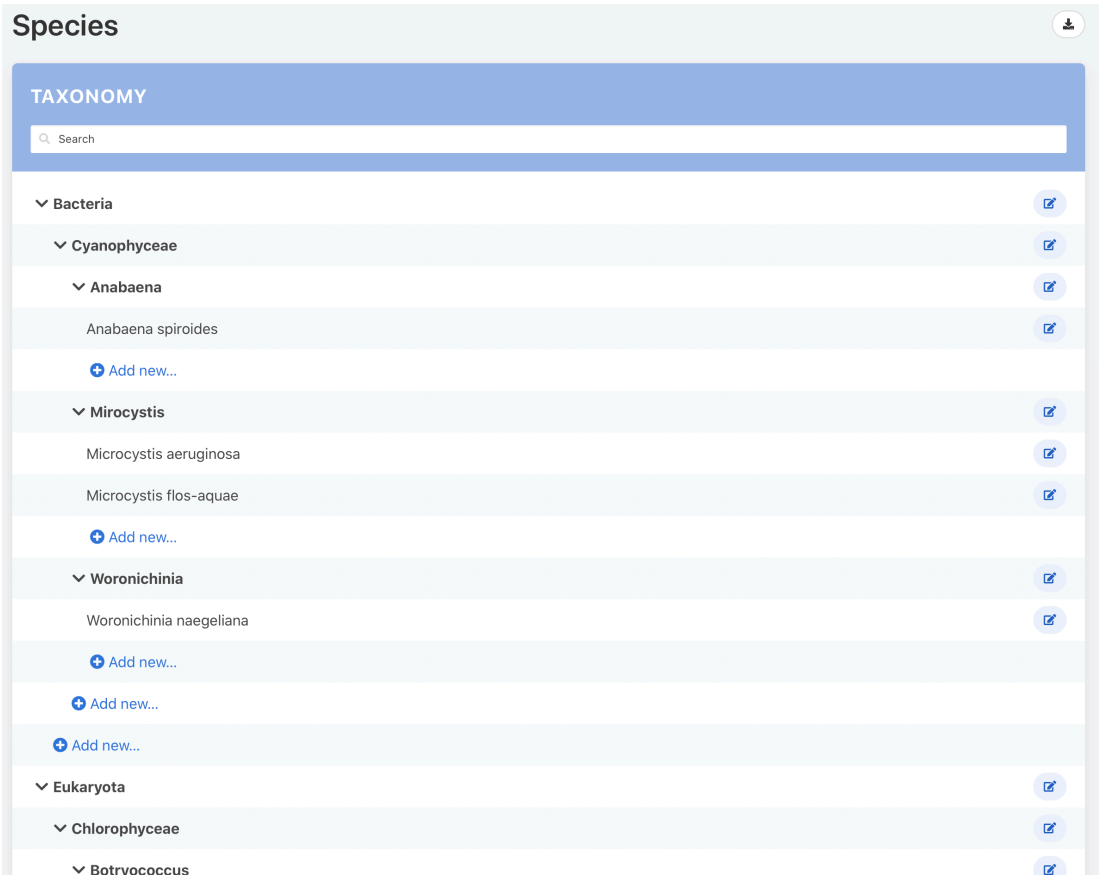

Figure 5.2: The Taxonomy viewer and editing interface.

Interface

Because our non-functional requirements explicitly ask for easy to use and correctly designed interfaces, we opted to use one of the most common design patterns when it comes to file upload: drop zones.

We decided to create a drop zone where managers could simply **drag and drop their files from their computers to the drop zones**, and uploading those files by doing just that. Of course, clicking the drop zone also brings the browser's native upload dialog, for those that prefer to upload that way.

To improve the quality of life of the UI it was also necessary to show the user the progress of every file being uploaded. This would help the user's experience while using the app and would help them to understand why they cannot still continue with the new sample creation.

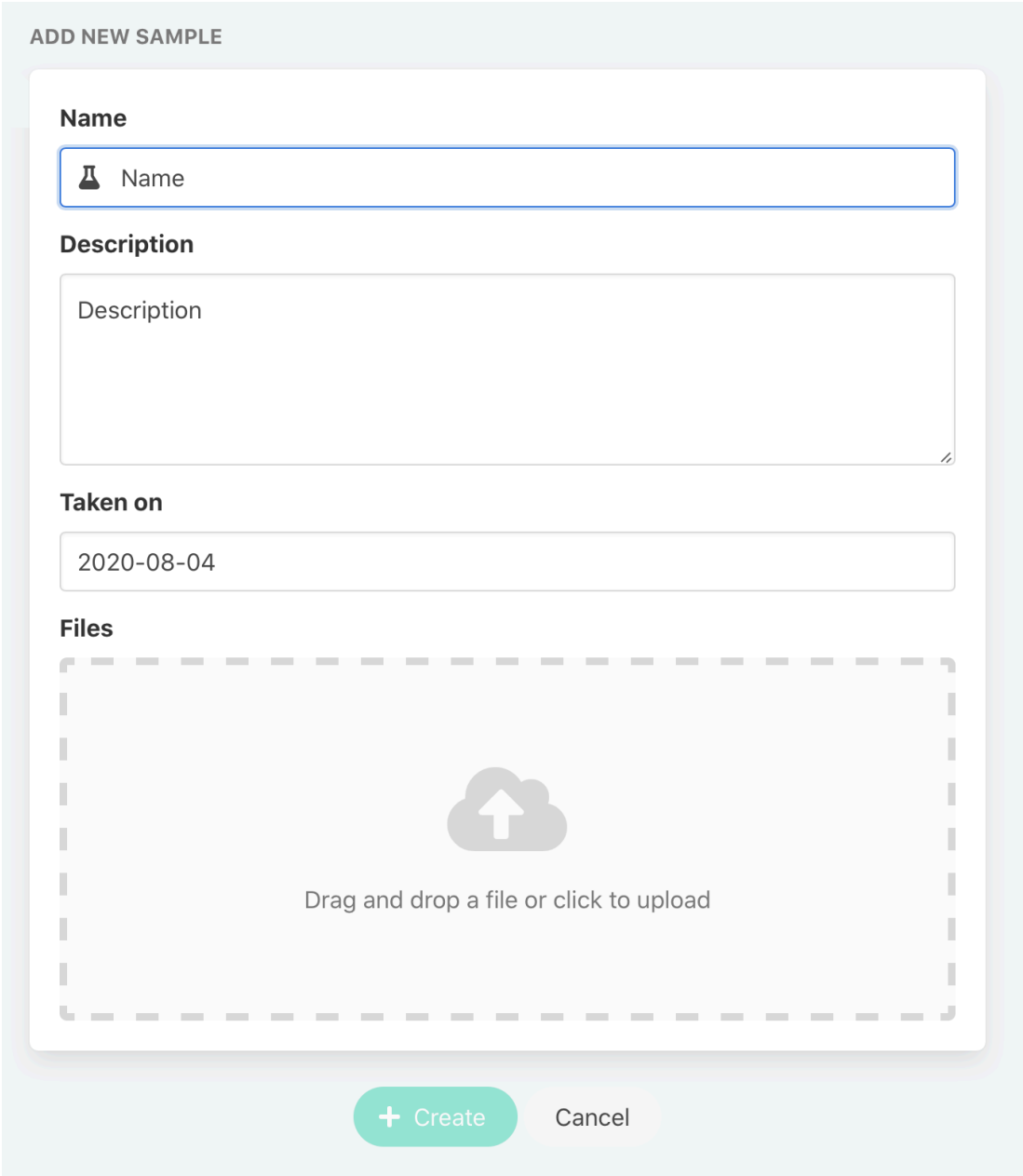

Figure 5.3: The upload sample form page.

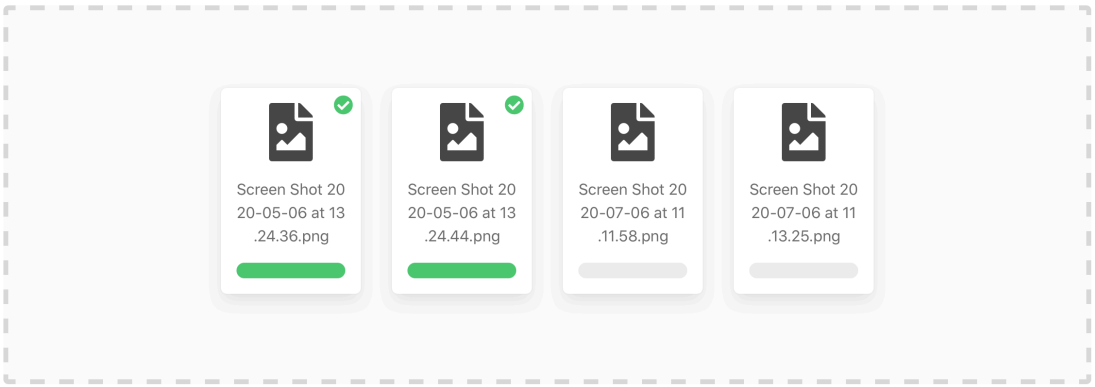

Figure 5.4: Image upload progress.

Garbage collection

To save time, images are instantly uploaded when they are dropped into the *drop zone*. This means that **a user can upload images without actually creating the sample**.

To avoid cases like this, the package used provides a tool to **remove old, orphaned files**. We simply configured it to run every hour and remove files that were orphaned for at least a day.

In order to schedule this task, `cron` was used. Cron is a scheduling tool for UNIX-like OSs that allows to schedule commands.

Compressed files upload

Because sample files are usually of high definition and quality, uploading them through the internet can be a very time consuming task. To aid with that, it was crucial to **support the upload of `.zip` and other compressed files** that intrinsically reduce the file size of the whole set of images. Please note that for simplicity reasons, we only implemented support for `zip` files. For other compressed file types, interfaces are only provided.

Depending on the tools used by the biologists to take the sample pictures, they can come with different directory structures. To simplify their workflow, we will be supporting the upload of images in compressed files, **independently of whether they are nested in multiple directories** or they are just images in the file's root directory.

Security risks

Uploading files can be very dangerous and open the application to multiple vulnerabilities. By default, all forms of the application already have input validation to prevent users from inputting incorrect data or even perform malicious code injections that are very common in incorrectly built PHP applications.

File validation is already provided by Laravel to check not only the extension of the files but its MIME type. However, **when uploading compressed files, validation must be done manually**.

The approach we used was to recursively navigate through all of the uploaded compressed file directories and **find images that satisfies the MIME types allowed** by the application. Those files that do not follow this requirements were instantly discarded and deleted.

Image processing

Because images were large in terms of file size, moving them to the application storage directory was not enough. These images were going to be distributed to the users through

their browsers and downloading high resolution images is also as cumbersome as is uploading them. Moreover, most of the images uploaded have `.tiff` extensions, which are **not supported by most common browsers**.

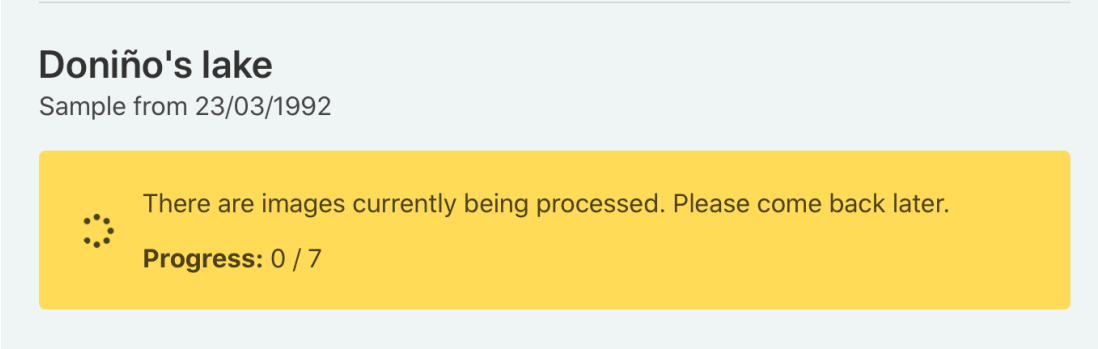

Figure 5.5: Warning that alerts that some images cannot be shown because they are being compressed.

We needed to process the uploaded images, to **compress and convert them to other commonly supported file types**, in order to avoid the previous problems. To do this, we used the queue. However, there was a problem. These tasks are very time consuming, so costly that it was necessary to show image placeholders while compression was being done.

This caused a problem in situations were there were multiple compression tasks in the queue and then another (for example, the sending email task for validation emails) was queued at the end. For the email to be send, **all preceding compression tasks would have to be run**.

Using a priority queue was not a valid solution either. Even though emails could be run before any compression task, there would always exist the possibility that a task could be currently running. In those cases, the email task would have to wait for that task to finish, and that could make emails take a little bit longer to send than expected.

The final solution was also a simplest one, to create another queue. In this case, **the new type of queue would only be tasked with compression jobs**. It also can benefit from the perks that queues have, like duplication, that can be great for scalability reasons.

Finally, as a result of all of the compression done, we ended up with the following images:

- **Original**: The original image that would be sent to automated services so that they take advantage of their high quality.
- **Compressed**: A compressed version in JPEG that will be used for biologist to identify phytoplankton.
- **Thumbnail**: A reduced version of the image in terms of dimensions. This version

would be used in places where the image would be shown in "aerial view" among lots of other images.

If the *compressed* versions were used instead, they could easily add up and end up supposing an even greater performance decrease due to the multiple HTTP request being sent with their bigger-than-necessary file size.

Chunk upload

Our last challenge regarding file upload was to chunk upload files. It was a requirement that any possible fault in the Internet connection would not interfere with the upload. This means that if for some reason we stopped at 50% upload, when we retried the upload we must be able to resume it from where we left off.

We used some libraries to accomplish this, with some work to support multiple requests and have both a custom interface and behavior. This is how it works:

1. When we drop a new file, it is chunked in the frontend to divide it into multiple pieces or chunks.
2. For each chunk, an HTTP GET request is sent to simply check if it has been already uploaded.
3. If the chunk was already uploaded (200 response code), it is discarded.
4. If it was not (204 response code), it gets uploaded as a common HTTP Multipart POST request. Chunks are stored in a temporal space of the application storage.
5. When the backend receives the last chunk upload (which does not necessarily has to be the last chunk), it sticks them all together and moves them to another part of the application storage used to keep the complete images.

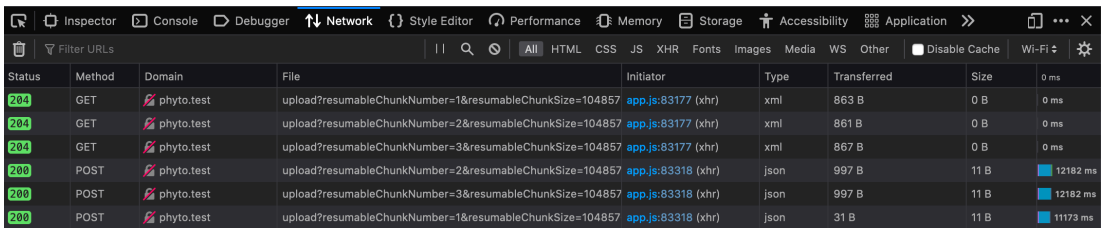

Figure 5.6: Screenshot of the request inspector.

In figure 5.6 a subset of the HTTP requests necessary for the upload is shown. These requests represent everything needed to upload three chunks, which is the amount of chunks set to upload simultaneously by default.

The first three requests ask the backend if those very same chunks have already been sent. These request send some metadata like the file name, its size, the chunk size, and many others, to make sure that the file and the chunk represent exactly the same file and chunk as some possible upload done previously. Because the file to be uploaded was never uploaded before, all three requests return a 204 No Content response.

```
/**
 * Handles the check request that Resumable.js does to ensure that
    a chunk hasn't
 * already been uploaded (for example, when the connection was
    interrupted or the
 * browser has been closed)
 *
 * @param Request $request
 * @return ResponseFactory|Response
 */
public function checkChunk(Request $request)
{
    $path = storage_path('app/'
        . config('chunk-upload.storage.chunks')
        . '/*' . $request->input('resumableIdentifier')
        . '.' . $request->input('resumableChunkNumber')
        . '.part');

    if (!File::glob($path)) {
        // Let resumable.js know that the chunk exists
        return response('ko', 204); // The chunk will be uploaded
    }

    return response('ok', 200); // The chunk will not be re-uploaded
}
```

Now, the frontend starts to upload those three chunks in good old multipart requests, which can be seen right after.

When the last chunk upload reaches the backend, it wraps them all together and moves them to the application storage with the following code.

```
/**
 * Handle file upload
 *
 * @param FileReceiver $receiver
 * @return ResponseFactory|JsonResponse|Response
 * @throws UploadMissingFileException
 */
public function upload(FileReceiver $receiver)
{
```

```
10
11      if ($receiver->isUploaded() === false) {
12          throw new UploadMissingFileException();
13      }
14
15      $save = $receiver->receive();
16
17      if ($save->isFinished()) {
18          return $this->saveFile($save->getFile());
19      }
20
21      $handler = $save->handler();
22
23      return response()->json([
24          'done' => $handler->getPercentageDone(),
25      ]);
26  }
```

Now the frontend form can continue as a common POST request where the file (or files) are just identifiers for the backend to know which file it is from its storage.

5.2.6 The Boxer

The development of the interface that would allow taggers to create bounding boxes is the core functionality of the application. As such, a lot of challenges were implied in its inception.

We will cover every single detail of the development of this UI that we will be calling the *Boxer* throughout the dissertation.

Interface

The interface should not only be able to create bounding boxes but also help the user to **navigate through the image** by zooming and moving around. These images are usually very big and sometimes zooming into a specific part can help with the identification of specimens.

A common UI that is a great host for all of these features is the **canvas editor UIs**. These are interfaces used by applications like Photoshop or MS Paint, which have a big area of the UI dedicated to a canvas where the content is displayed, and the rest of the real state is used to show toolboxes with the different tools the application provides.

We decided to use this kind of interface for all our bounding box creation and then tagging (every bounding box will be tagged with a specific identified species).

In figure 5.7 the interface of the Boxer is shown. Here is the description of each of the components identified by a number:

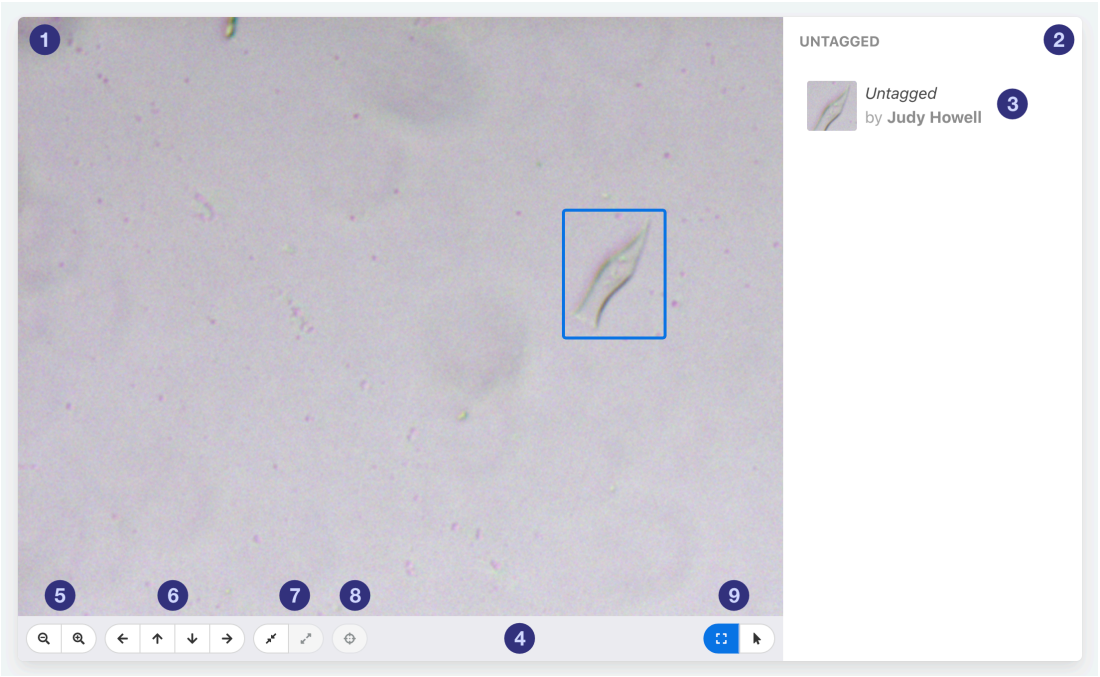

Figure 5.7: The interface of the Boxer.

1. **Canvas**: The container of the image. This area lets you interact with the image depending on which mode the Boxer is in.
2. **Sidebar**: The sidebar shows a list of all bounding boxes, separated into categories depending on whether they have been tagged (identified with a specific species).
3. **Bounding box**: Each bounding box created is shown in the sidebar, along with a miniature of the selected area to quickly identify it. They are also drawn in the canvas with a blue outline that turns yellow when you hover the corresponding bounding box in the sidebar, to quickly identify which one is it.
4. **Toolbox**: The container of all tools that can be used to interact with the image.
5. **Zooming tools**: These tools allow you to zoom in and out in the image.
6. **Navigation tools**: These tools allow you to move around the image with precision.
7. **Expanding tools**: A tool to maximize the image to their original size and another tool to resize to fit the available space.
8. **Reset tool**: A tool to recenter the image, leaving it in its original position.
9. **Mode switcher**: Switches between the different modes available in the Boxer. As of Sprint 4, the following modes are available:

(a) **Tagging mode**: The one that is selected on the screenshot. When selected, you can draw bounding boxes by clicking and dragging anywhere on the canvas.

(b) **Move mode**: When this mode is enabled, clicking and dragging on the canvas lets you move around the image. You can also use your mouse wheel to zoom in and out.

Implementation

Making the previous interface real was very tricky. For starters, there was not any third party package that we could base it on. Every third party library of the sorts either was imagined as an image editor or were not customizable enough to make room for all the features we intended to ship.

We finally decided to **implement the UI from scratch**, using React to build a clean and reusable component. However, as the development of the UI was evolving and more features were added, React soon started to force us to develop unhealthy code.

Because React's components state is not shared among components, we had to manually pass it between them. This created unnecessarily complex code that did not help its maintainability or even legibility. To overcome this problem, we decided to **introduce Redux into the project**, to manage all of the Boxer's state in one place. Thanks to that, the code used was greatly reduced and adding more features in future sprints was made simpler.

Nonetheless, the development team lost a lot of time getting to know this new technology because they have never used it before. As consequence, some features intended for the current sprint, such as the ability to edit and remove bounding boxes could not be done and **were postponed to the next sprint**, with the corresponding creation of new user stories to be added back into the PB.

Live interaction

Losing any job done in the Boxer while interacting with it cannot be allowed, as it is very valuable. To avoid any possible data loss, we decided to use a live interaction pattern. This means that every time a bounding box is created or modified, **an AJAX request is sent to the backend to synchronize the changes** and, hence, avoid any possible data loss.

To let users now when a bounding box is not synchronized with the backend, a spinning loading icon is shown next to it to avoid confusion.

Drawing and moving by dragging

Implementing the ability to use the mouse to draw a bounding box or to move around an image took away most of the time available for the sprint. To accomplish this, we had to make use

of multiple native JavaScript events. For example, this is how the creation of bounding boxes works behind the scenes:

1. A `div` element contains the image as the background and will be the target of all of the JS events used.

2. When a user clicks and holds, the `mouseDown` event is fired.

   The event will save a starting point for the `(x,y)` coordinates of the bounding box. These values are retrieved using the `offsetX` and `offsetY` properties of the event object, which represents the horizontal and vertical (respectively) offset from the elements padding edge (i.e. if we happened to click exactly on the top left corner of the image, these values would be `(0,0)`).

3. Then, thanks to the previous event, we will start taking into account any `mouseMove` fired. We use this event because a little HUD is shown to understand the dimensions of the bounding box we are creating, so it needs to constantly be updated to reflect the real size of the final bounding box.

4. Finally, when we have the desired size for our bounding box, we end our clicking action. The `mouseUp` event will be fired. We will use this event object's property `offsetX` and `offsetY` properties to know the exact position of the cursor at this time.

   However, because the backend stores bounding boxes as sets of the original starting point coordinates `(x,y)` and then their `width` and `height`, it is necessary to subtract the final offset, to the offsets of the original point.

The previous steps summarize pretty well how the creation of a bounding box process works in the frontend. However, it is worth noting that there was a lot of more challenges with just a *tiny* part of a UI.

Implementing the ability to create a bounding box upwards and to the left with respect the origin point, to have bounding boxes on top of the image while being able to interact with it or zooming, and to synchronize the size of the bounding boxes while zooming were some of the most notable challenges that we are not going to go into detail about.

Showing options

We had to think of a way to show the options to remove a specific bounding box or to edit it. Initially, the easiest way would be to add new options next to each bounding box in the sidebar. Despite that, we thought that would not follow the user's workflow, because the sidebar purpose is to have an overall look at the identified elements of an image. The actual

bounding boxes are in the canvas, so it felt natural that **we should be able to interact with them in the canvas**.

As our first approach, we decided to make options appear when clicking on a specific bounding box. Soon, however, we realized that his was not something easy to do. We spent a lot of time trying to make a way to differentiate the dragging action from just a click on a bounding box. Because bounding boxes were elements on top of the image, we had to disable any possible events to be able to receive the dragging events in the image underneath. This makes it very difficult to find a way to both actions to coexist.

We did not want to get to the deadline of the sprint without implementing everything like the previous sprint (Sprint 4), so the amount of time we could spend trying to find a way to do it was limited. In the end, we did not find a solution, so we decided to go with a less nicer approach: we introduced an **editing mode**, where you could only **click on bounding boxes to show their editing options**.

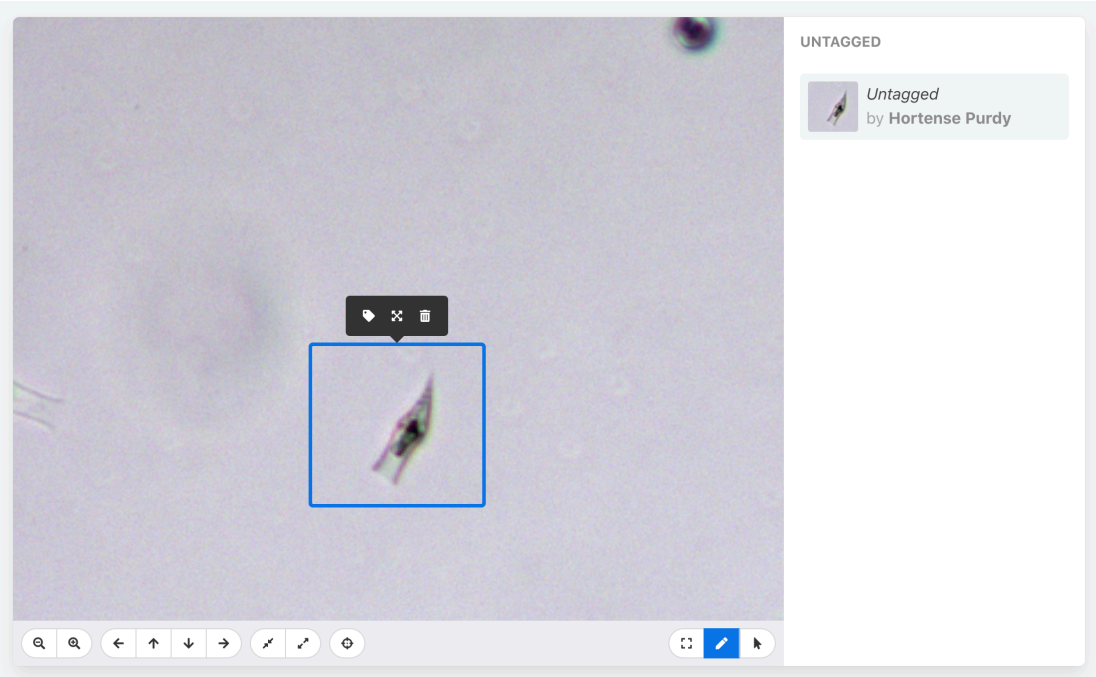

Figure 5.8: The edit mode showing the options for a bounding box after clicking on it.

Editing

Just like when creating a bounding box, editing them became something more difficult than anticipated. Because editing and creating rely on different ways to do it, we had to create from scratch a new UI.

The usual way to edit rectangles in photography edition tools is by providing resizing

nodes in each corner that you can move around to readjust the dimensions of the rectangle. This interface is very well-known, so we used it to edit our bounding boxes without second thoughts.

To implement it we reused part of the *selection* when creating a bounding box, but it was required to add specific cases for every one of the edge nodes, which behave differently depending on their position.

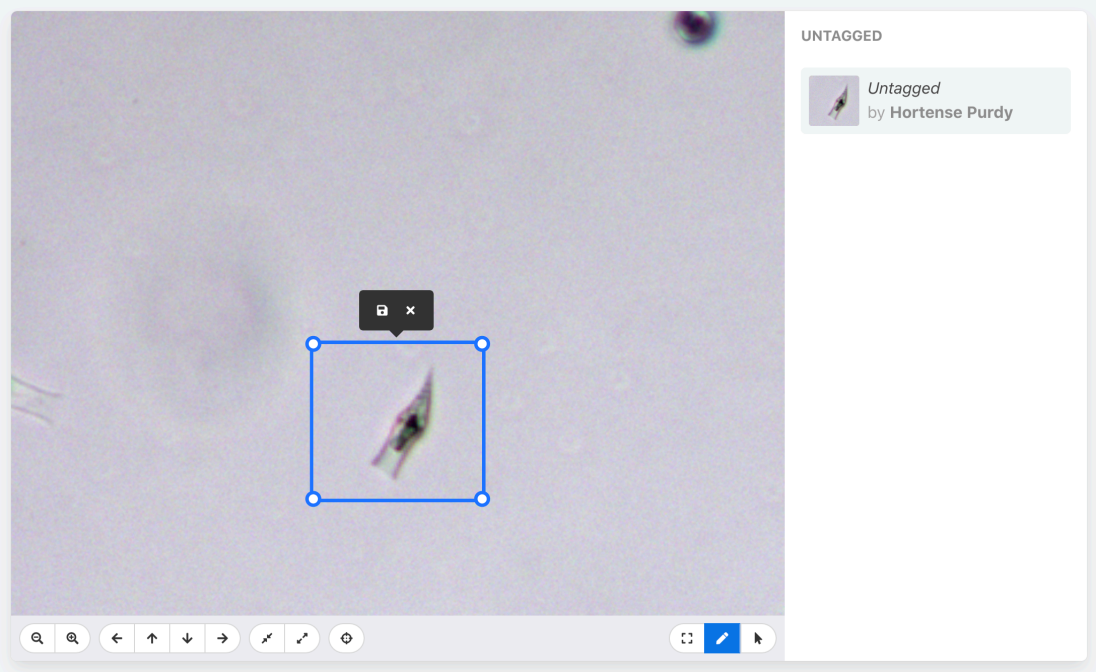

Figure 5.9: The editing interface with options to save or to restore the original size.

There was, nonetheless, an interesting problem when it came to the implementation of the node edges. When clicking a node, the dragging process starts, but the `mouseMove` event is *listened* on the image. Because JavaScript has event bubbling, the `target` element of the event does not necessarily need to be the element where the event is listened. An event generated in a target element will *bubble up* until it finds an element that has a listener for that event.

In our particular case, because the edge is moving with the mouse after clicking, the `target` of the `mouseMove` event is actually the node itself. This interfered with the retrieval of the `offsetX` and `offsetY` properties, that would now be relative to the node, instead of the image, driving them completely useless.

To fix it, we used the `user-events` CSS property to disable events on the edges, but because we need to be able to click them to start the dragging process, we used JavaScript to apply this property only when the edge was already clicked on, and a drag is in process.

Scrolling to a bounding box

When an image has a lot of species, the amount of bounding boxes could make the interface to show a scroll bar. This can make it even more difficult to identify which bounding box is which in the sidebar. In the previous sprint we already implemented a way to tell where a sidebar bounding box is placed in the canvas by changing its color once you hover on it on the sidebar.

However, it was not possible to identify a bounding box from the canvas in the sidebar. To do it, we changed the background color of the sidebar bounding box when when click a bounding box in edit mode, as shown in figure 5.9.

In those cases where the highlighted bounding box was in the scroll's not visible area, we had to automatically scroll the container to make it visible. To accomplish this, we took advantage of the `scrollTop` property which represents how many pixels from the top are scrolled. The following snippet explains the logic to know when the scrolling should be done.

```
useEffect(() => {
    if (!box.focused) return

    const element = ref.current
    const parent = ref.current.parentNode

    if (parent.scrollTop > element.offsetTop || (parent.scrollTop +
    parent.offsetHeight) <= element.offsetTop) {
        parent.scrollTop = element.offsetTop - 10
    }

}, [box])
```

Finally, to make it even more user friendly, we added an animation with the new CSS property `scroll-behavior` and its value `smooth`, which automatically selects an animation function according to the user agent.

Thumbnail in the sidebar

The team thought it would be a good idea to create a thumbnail of the bounding box to have at a glance an understanding of what does it contain. However, at the moment, bounding boxes were only stored as their coordinates and dimensions. It was not ideal that for a thumbnail to be shown, we would have to crop the image into new thumbnails that are stored with every bounding box.

To keep the storage usage low, we decided to create those thumbnails directly on the frontend, programmatically. With the help of CSS and JavaScript, we used the same image in the canvas, resized it with CSS `transform` and placed the image in the right position

using a simple property like `background-position`.

Just like that we were able to show useful thumbnails with no performance overhead and, most importantly, saving space in the backend.

5.2.7 Workload balancing

Making real the assignments was practically the most relevant thing done in Sprint 6. It was necessary to create an algorithm that would create assignments that always followed the aforementioned rules.

The approach used was to:

1. Compute the availability of every assignee involved for each image.
2. Evenly assign, where possible, those assignees that participated in previous compatible tasks.
3. Assign now all new assignees, evenly.

5.2.8 Automated Services

Integration

Automated services are their own programs that may be running somewhere in the Internet or in our own local network. It was necessary to find a way to communicate between those services and our platform, so that assignments can be sent and their response could be received with the resulting bounding boxes.

We used a REST API to solve this problem. Every assignment would be sent to the corresponding service in an HTTP Multipart POST request with:

- The **original image** that the service must work on.
- A **callback URL** that the result must be sent to.

Because the work of the automated services is not instant, they must proceed with their work asynchronously. Thus, it was necessary to provide a webhook to which they could send back the species found.

This webhook enjoys all security measures that the activation link has: it is signed and it is uniquely created so that every request can answer to only one URL, guaranteeing that no automated service can provide results to any other unauthorized assignment, and preventing any possible errors in the service's end, that could send back the result to the wrong assignment.

Of course, because sending an HTTP request is also a time consuming job, we asynchronously run it with our queue. Here is a snippet of how the HTTP request is sent:

```
/**
 * Execute the job.
 *
 * @return void
 */
public function handle()
{
    $service = (object) config('automated_identification.services.'
        .$this->assignment->service);

    Ztp::asMultipart()->post($service->endpoint, [
        [
            'name' => 'callback',
            'contents' =>
    URL::signedRoute('automated_services.receive_bounding_boxes',
                ['assignment' => $this->assignment]),
        ],
        [
            'name' => 'image',
            'contents' =>
    Storage::get('public/'.$this->assignment->image->original_path),
            'filename' =>
    basename($this->assignment->image->original_path),
        ],
    ]);

}
```

Species between applications

We also needed to provide a way to communicate species between both applications. How would an automated service communicate to our platform that it found a *Thalassionema nitzschioides*?

To prevent problems with naming (a species might be named in a different way in the application that its real name is, for example) we decided to go with the easiest approach: to use the platform's unique identifiers for every species.

In order for automated service developers to know the identifiers for every species, we had to provide the possibility to **download the taxonomy database**. We used a common format, JSON, to create the file that would contain the information. This file would only contain the necessary information, and we would also take care that it would be correctly formatted to

make sure that it is easily readable by a human.

Testing

In order to test this functionality we would need to make use of some automated service already built. However, at the moment there were none available because **we would also have to create their REST interface**, which was completely out of the scope of this project.

To be able to test the functionality, we implemented a **dummy service that would create random bounding boxes**. This service was implemented using Node.js because of how easy it is to build REST APIs with it. Because JavaScript is an event-driven asynchronous language, we used `setTimeout` to delay the response of the automated service to make it feel more real.

5.3 Testing

For every implemented feature in each sprint, tests were created. Due to the nature of unit testing and the amount of integration most components have in the application, it was necessary to mock calls to other modules to ensure tests are unitary.

Thankfully, most of the default modules Laravel brings already support mocking out of the box. Nonetheless, a lot of our custom build modules, required to be manually mocked.

Here is a snippet of code that tests the creation of samples.

```
/**
 * @test Test the addition of a sample to an already created project
 */
public function add_sample_to_project()
{
    Queue::fake();

    $files = collect(range(1, rand(1, 10)))->map(function () {
        return $this->faker->word . '.' .
    $this->faker->lexify('???');
    });

    $this->mock(FileUtils::class, function (MockInterface $mock)
    use ($files) {
        $mock->shouldReceive('storeImages')->withAnyArgs()
            ->andReturn($files);
    });

    $projectService = $this->app->make(ProjectService::class);

    $project = factory(Project::class)->create();
```

```
20
21      $name = $this->faker->sentence(5);
22      $description = $this->faker->paragraph(2);
23
24      $projectService->addSampleToProject($name, $description, new
        Carbon, $files, $project);
25
26      $this->assertEquals(1, $project->samples()->count());
27
28      $projectImages = $project->samples()->first()->images;
29      $this->assertEquals($files->count(), $projectImages->count());
30
31      foreach ($files as $file) {
32          $this->assertTrue($projectImages->contains(function (Image
        $image) use ($file) {
33              return $image->original_path == $file;
34          }));
35      }
36
37      Queue::assertPushed(NormalizeImagePreviewJob::class,
        $files->count());
38  }
```

As we already mentioned in section 5.2.5, adding a sample (and its images) implies the firing of multiple compression tasks. These tasks must be mocked to avoid actually running them in our test. Because queues are supported out of the box by Laravel, we could easily mock it with the facade method `fake`.

Moreover, to upload files we had to implement our own service, called `FileUtils`, which makes sure of storing images in the app's configured storage and returns the final path where the images can be found, among other things. This should also be mocked to **avoid actually storing any kind of file in our tests**, which are out of the scope of the test's objectives altogether. To mock this, however, we had to manually do it ourselves, as seen in line 12.

Despite all of that, mocking does not mean that we completely forget about this necessary interaction, so we make sure that the queue would receive the according jobs, as seen in line 37.

5.4 Improving testing quality

Unless following a Test Driving Development (TDD) workflow (which we did not follow for this project), tests are usually the first thing to leave behind when there is not much time left.

For this, we needed to make sure that in the case that we lacked test **quantity**, that would

not harm test **quality** (quality over quantity). We used some techniques to ensure that:

5.4.1 Random Data Generation

This technique allows to randomly generate data that, usually, is aimed at testing edge cases. For example, a function that test if a number is greater than 3 would have the surrounding values as edge cases that are usually prone to error. Random data generation helps avoiding this kind of mistakes.

In line 66 of the previous snippet we can see how we use the Faker library to create random file names to be used in the mocked FileUtils file. We also used random sizing for the amount of files to continue with the practice.

In line 19 we also used Faker, but under what is called a factory, which is a function that creates random instances of our models. This ensures that our models are as random as possible and increases the quality of the tests done.

With every run of the tests, you make sure that more edge cases are being tested with your code.

5.4.2 Mutation Testing

Mutation Testing is a technique based on **changing your application's own code** with the intention that your tests would break, hence proving that they can detect those faults.

We used a library called Infection, which already supported our PHP unit testing framework, PHPUnit. With Infection, we make sure that typical mistakes such as common programming errors or specific PHP errors do not happen in our codebase.

Here is the output that running infection returns.

```
1  You are running Infection with Xdebug enabled.
2
3      ____      ____           __  _
4     /  _/___  / __/__  _____/ /_(_)___  ____
5     / // __ \/ /_/ _ \/ ___/ __/ / __ \/ __ \
6   _/ // / / / __/  __/ /__/ /_/ / /_/ / / / /
7  /___/_/ /_/_/  \___/\___/\__/_/\____/_/ /_/
8
9  Infection - PHP Mutation Testing Framework
       0.15.0@1d79de53cbc1dfa902e1ea4afba6c3f0214d5224
10
11 Running initial test suite...
12
13 PHPUnit version: 8.5.0
14
15    26 [============================] 46 secs
```

```
16
17 Generate mutants...
18
19 Processing source code files: 6/6
20 Creating mutated files and processes: 42/42
21 .: killed, M: escaped, S: uncovered, E: fatal error, T: timed out
22
23 ......................M..............MM...           (42 / 42)
24
25 42 mutations were generated:
26       39 mutants were killed
27        0 mutants were not covered by tests
28        3 covered mutants were not detected
29        0 errors were encountered
30        0 time outs were encountered
31
32 Metrics:
33          Mutation Score Indicator (MSI): 92%
34          Mutation Code Coverage: 100%
35          Covered Code MSI: 92%
36
37 Please note that some mutants will inevitably be harmless (i.e.
       false positives).
38
39 Time: 1m 2s. Memory: 18.00MB
```

5.4.3 Static Analysis

To ensure the quality of our code, we used our own IDE's static analysis tools to enforce a code style that follows the specific Laravel deviation from the official PHP PSR-1 and PSR-2 style standards.

5.4.4 Performance Testing

While developing, we constantly made use of the tool LaravelDebugBar to detect performance problems at runtime to be able to avoid them. This was very useful when dealing with lots of SQL queries, which helped to visualize the SQL queries run, making easier the detection of n+1 problems.

Apart from showing the SQL queries run, it also allows us to see the compiled views for every request, chained requests and even the amount of memory being used, among many other things.

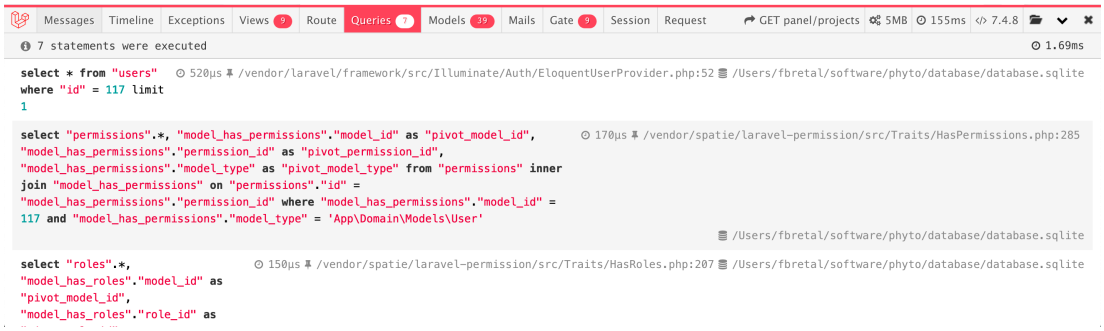

Figure 5.10: Laravel DebugBar showing the SQL run for the current request.

5.5 Data Model

For the application's design two types of models (how entities are called in Laravel's world) were considered:

- **Domain Models**: These are the models that are required by the application's domain (Samples, Species, Projects, etc.).
- **Application Models**: These models are needed for the application to work. For example, queues make use of the Job model to store all pending jobs.

Figures 5.11 and 5.12 contain some diagrams for both type of models. Please take into consideration that these were generated from a SQLite database, so types are extremely simplified, but it helps to grasp the general idea of the domain of the application.

5.6 Service Diagram

Finally, we will have an overview of the main services of the application and their methods in figure 5.13. Please take into account that because of the nature of Laravel's architecture, this will not cover every single detail of the app's functionality.

Figure 5.11: The domain of the application.

Figure 5.12: Application-related models.

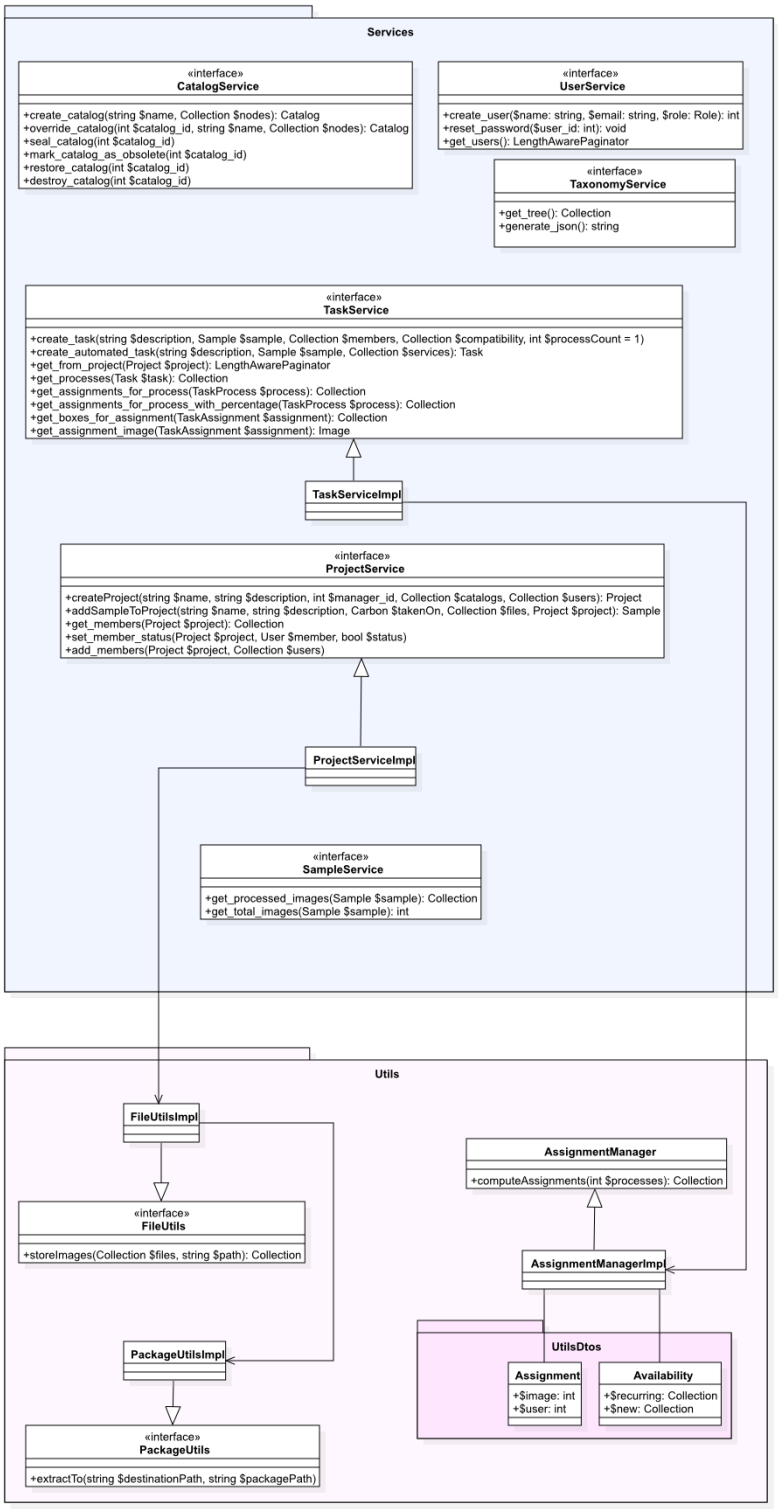

Figure 5.13: The main architecture of the application's services.

Chapter 6

Final Product

To wrap up, we will be using this chapter to show an overview of the final product. Please take into consideration that we will not be covering every detail of the application, and that some uninteresting sections might be left behind, like as simple forms and such.

6.1 The Administration Panel

As we mentioned in previous chapters, the administration panel aims to be the central management section which will provide the necessary tools for project managers and higher ranks. It can be accessed through the corresponding button in the navigation bar, right next to the button that shows the logged in user.

It is composed by a sidebar that lets you navigate through all the management modules. This sidebar also contains a list of all of your managed projects (if you have any) so you can access to them faster.

6.1.1 Project Management

The project management module is the default module shown when you enter the administration panel. A project manager can see in this section all of their managed projects, but higher ranks can use it to see an overview of all projects in the application, as seen in figure 6.1.

For each project, we get links to its management section within the panel, and a shortcut to the same project page that taggers would see from their accounts.

6.1.2 Managing a project

When managing a project we are shown the general view (figure 6.2), which shows some statistics about the project itself, along with some basic metadata. Right to the title, there is another shortcut button to access to the project's view for the taggers.

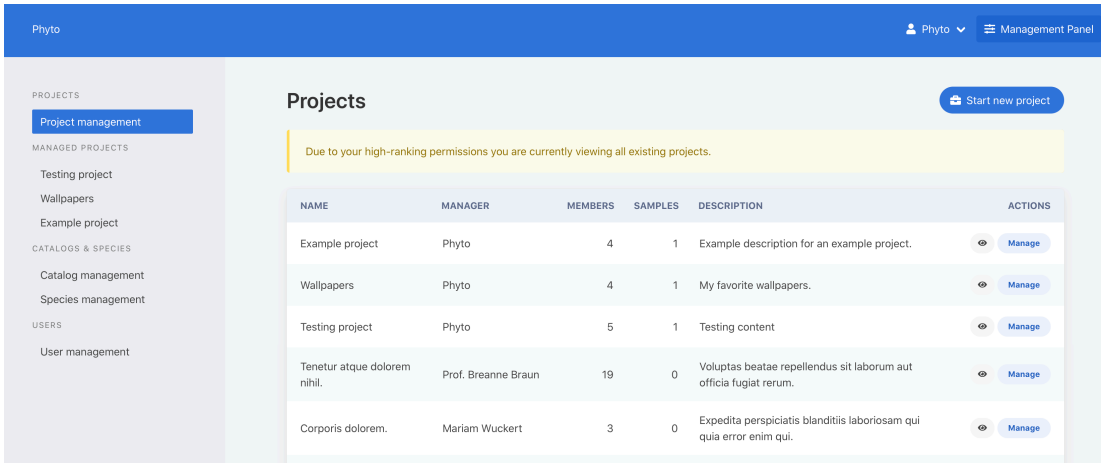

Figure 6.1: The project management viewed as an administrator. Because of our role, we get a warning that tells us we are seeing all of the application's projects.

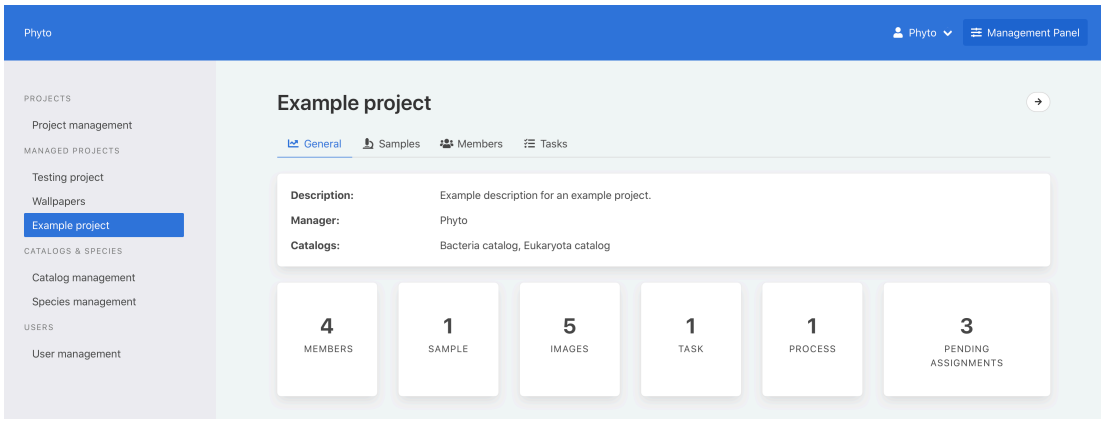

Figure 6.2: Managing section for a specific project.

We are also provided with multiple tabs to access the sampling managing section, the member list and the task manager.

Samples

The sample managing section (figure 6.3) lets you upload a new sample with all of its corresponding images as previously shown in figure 5.3. It also lets you view any possible identification processes started for that sample (figure 6.8). An overview page is also provided to see all the images of the sample in a gallery-like style (figure 6.4).

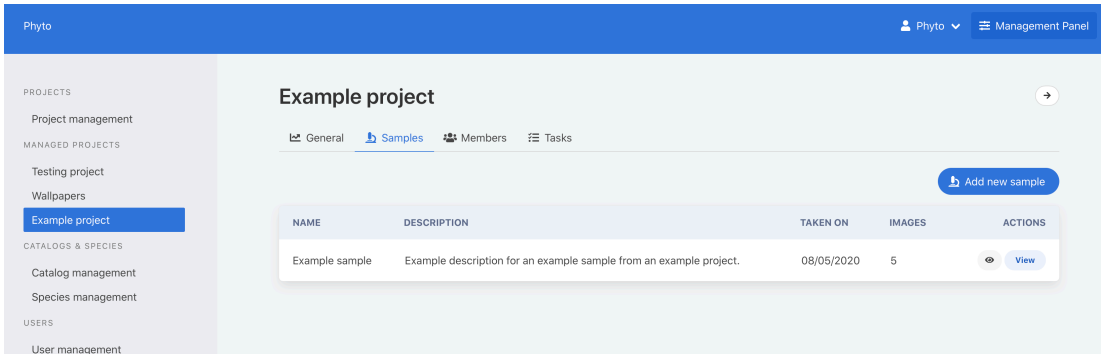

Figure 6.3: Sample managing section for a specific project.

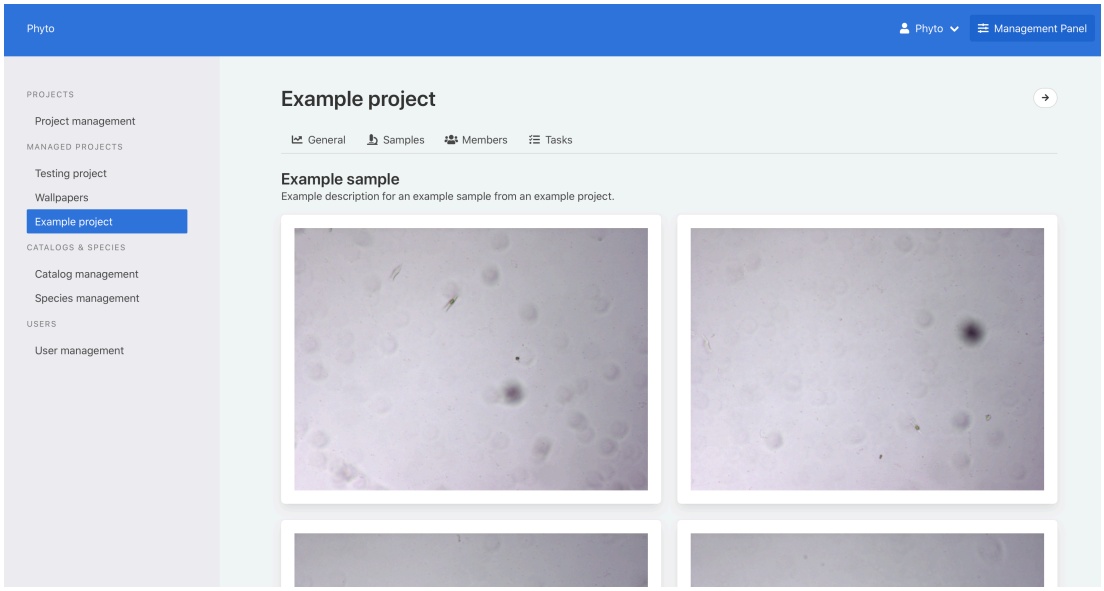

Figure 6.4: Overview of the images of a sample.

Members

In the member list (figure 6.5) we will see all the members of our project. We are also given the opportunity to disable (not remove!) a user so it does not participate in the project anymore.

To add a user we provide a nice selector that searches the user database efficiently with AJAX (figure 6.6).

6.1.3 Tasks

The task manager (figure 6.7) shows all tasks of a project with their corresponding status and progress, among some metadata. Just like with projects, we are given the opportunity to see a list of all identification processes that it has started (figure 6.8).

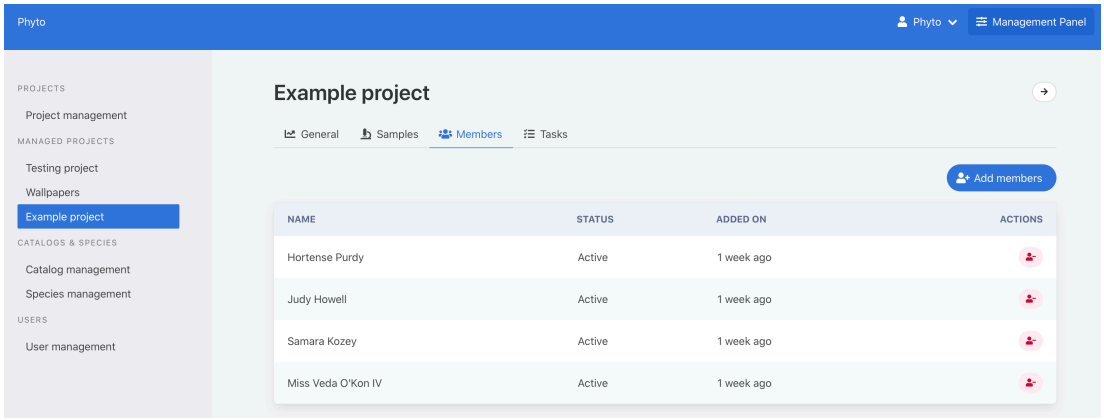

Figure 6.5: Member list.

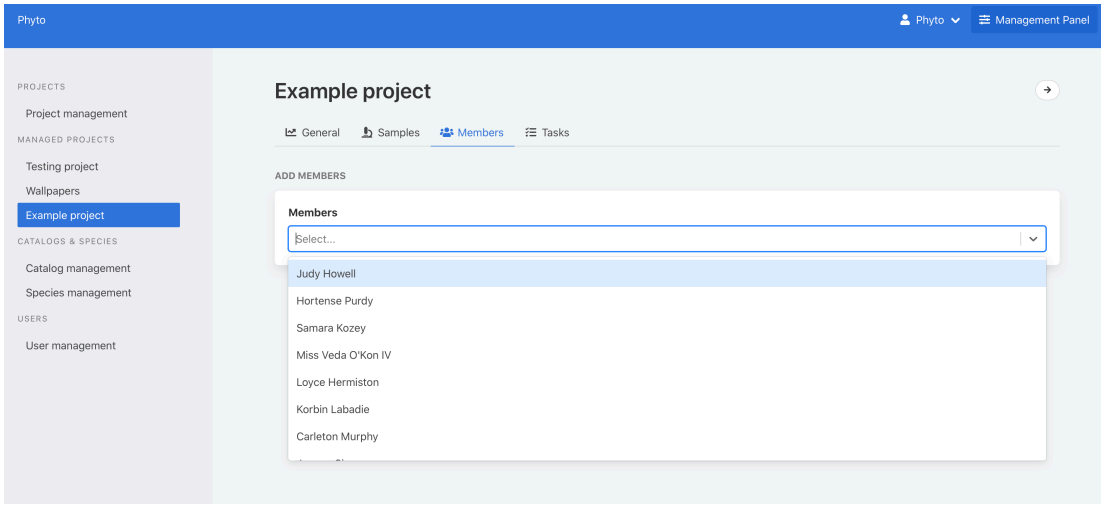

Figure 6.6: Add a new member form.

6.2 Catalog Management

The catalog management module (figure 6.9) shows an overview of all catalogs for every user that has the role supervisor or higher. Every catalog has different actions available on the right side depending on their state.

For "editing" catalogs (i.e. catalogs that have just been created and are not available for use) you can edit them, completely delete them or seal them. Sealing a catalog changes it state to "sealed" which make them inmutable.

Seal catalogs are those that can be assigned for projects to use as their source of species. These catalogs can only be marked as "obsolete". Then, obsolete catalogs, can only be restored into sealed catalogs to recover them.

Independently of the state, you can always start a new catalog from a new one (clone a

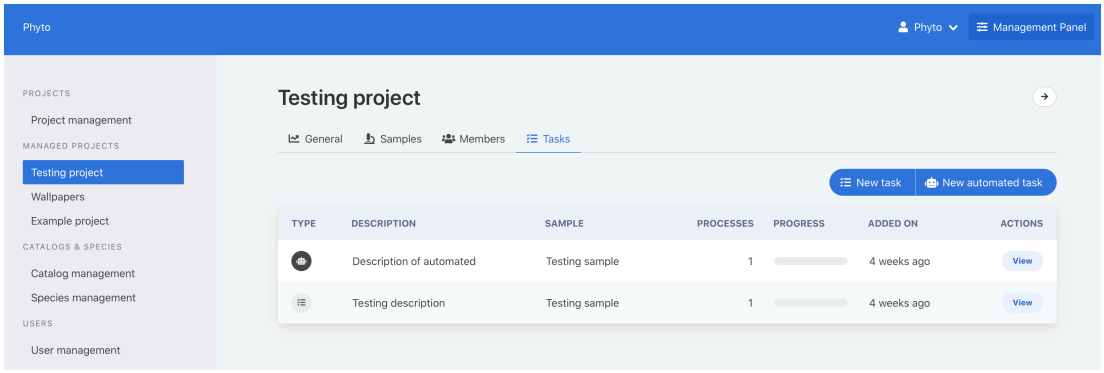

Figure 6.7: The task manager.

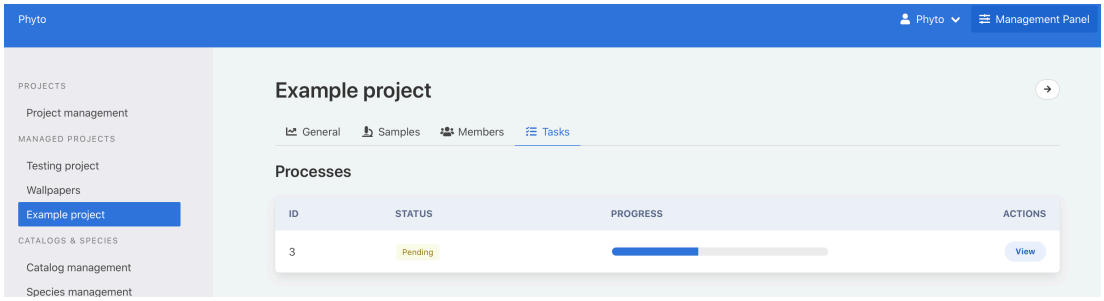

Figure 6.8: The process list. This view is shared between tasks and processes.

catalog). That would create a new catalog that would always start as "editing".

6.3 Species management

The species management module (figure 5.2) lets supervisors (and higher) see at a glance all of the taxonomy database available in the application. In this section you set the species and their containing groups that can be available to use in catalogs. From here you can also change a name of a node (a species or any of the containing groups) or create new nodes.

For those situations were the taxonomy is very big, you can filter the hierarchy in real time by typing in the "search" field.

On the top right of figure 5.2 we can also see a button to download the JSON file with all the taxonomy database.

6.4 User management

The user management module allows administrators to manage the user pool of the application. They can use this section to add new users, but can also use it to reset them, which allows the user to change their password if they have forgotten them.

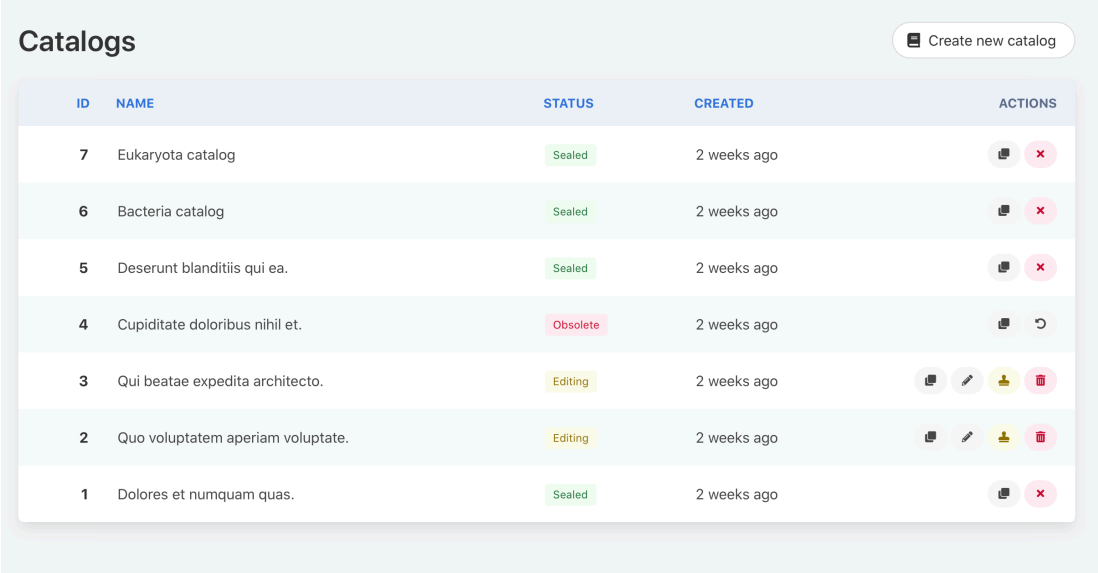

Figure 6.9: The catalog management page.

Every user is giving an automated avatar generated by hashing their name so that they can be easily identifiable.

6.5 Dashboard

The main page of the platform is known as the dashboard (figure 6.10). Here is where you can see the work you have left in the application. It's mainly focused on taggers, but everyone can see it.

It is divided in three sections. On the right, you can see your assignments. These are shown randomly from your total assignment list from different projects and processes, so if you refresh the page, the assignments would change.

On the left you get a list of all of the projects you're part of and all the identification process you're involved in. For each one of them you also are given a notification of your pending work on each one of them.

If you have administration access, you are also given some shortcuts to the different modules of the administration panel on the top.

6.6 Project

Every project has its main project view. This is the actual project that taggers will see, the project view that is not related to management. A project view comes with a gallery view of a potpourri of all the images in that project, just because aesthetics™ (figure 6.11).

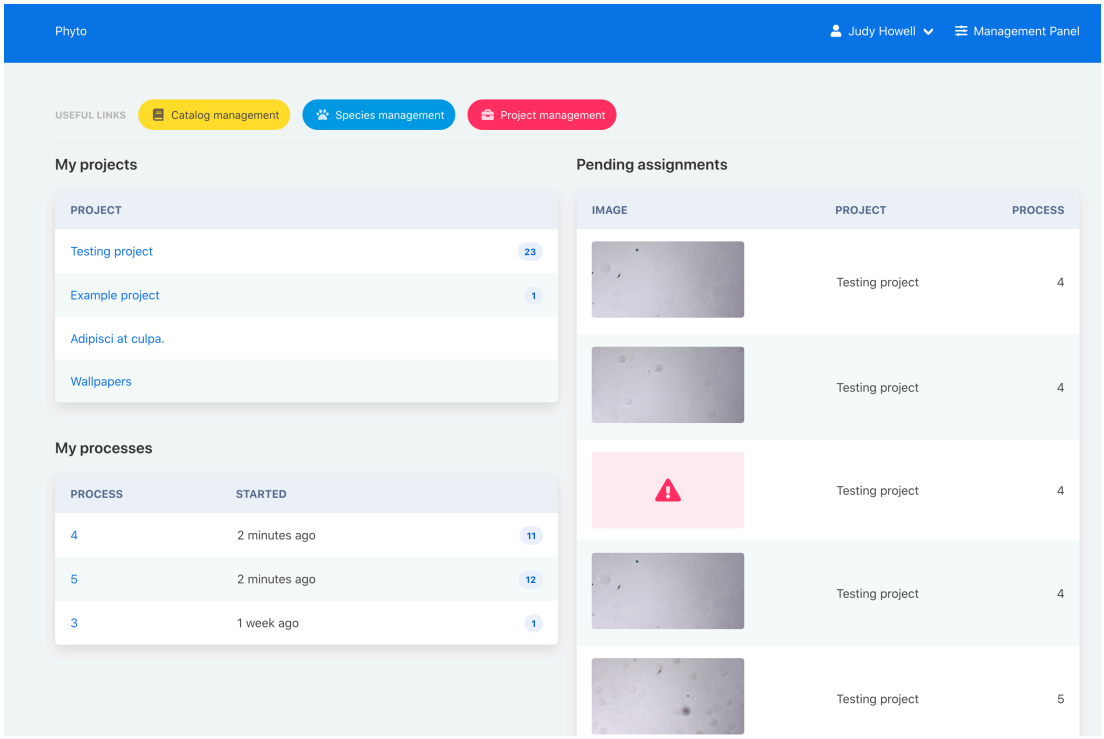

Figure 6.10: The dashboard. In the random task list in the right it is shown how one image doesn't have a thumbnail because its corresponding compression task has not run yet.

6.6.1 Assignment list

Of course, for every project you have a list of all your assignments (figure 6.12), with a notification of how many assignments you have pending. There's also a filter to show only assignments for a specific process. Every assignment lets you access the tagging page.

Tagging

The tagging page (figure 6.13) is the core of the application. Here the boxer is shown so that taggers can create bounding boxes and identify the species they have found.

On the bottom, there is a pagination with thumbnails of other assignments following the current one you are viewing.

There's also a button to set the assignment as finished so it cannot be edited anymore. Of course, a finished assignment changes its Boxer UI to the non-edit version.

6.6.2 Member list

Finally, any member can see the rest of the members of a project (figure 6.14). The member list has a nice little identifier to highlight yourself in that list.

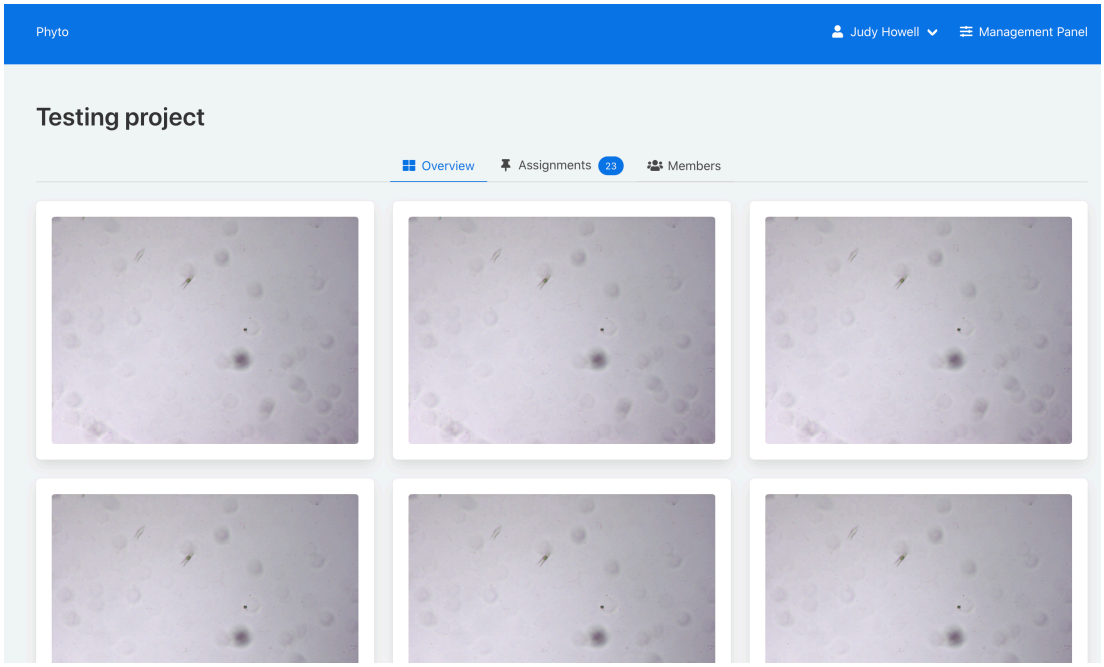

Figure 6.11: The main view of a project for taggers.

This view also works as a hall-of-fame (or of shame, depending on your work done) that shows your assignments done and those you have not yet finished.

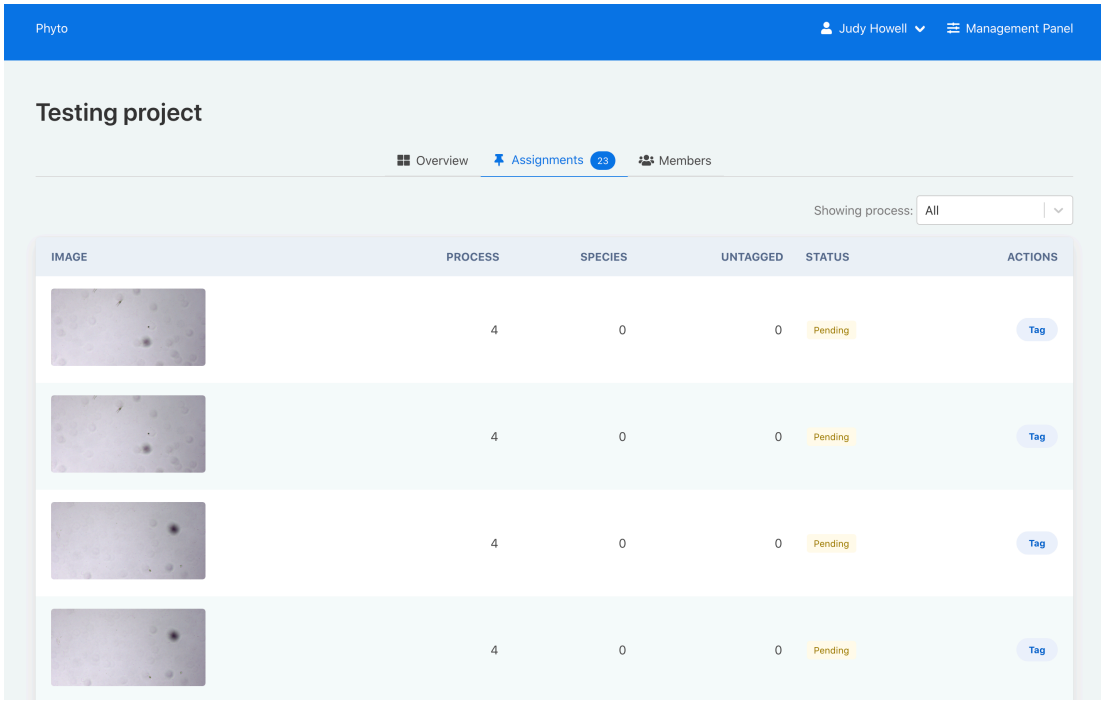

Figure 6.12: The assignments view for a project.

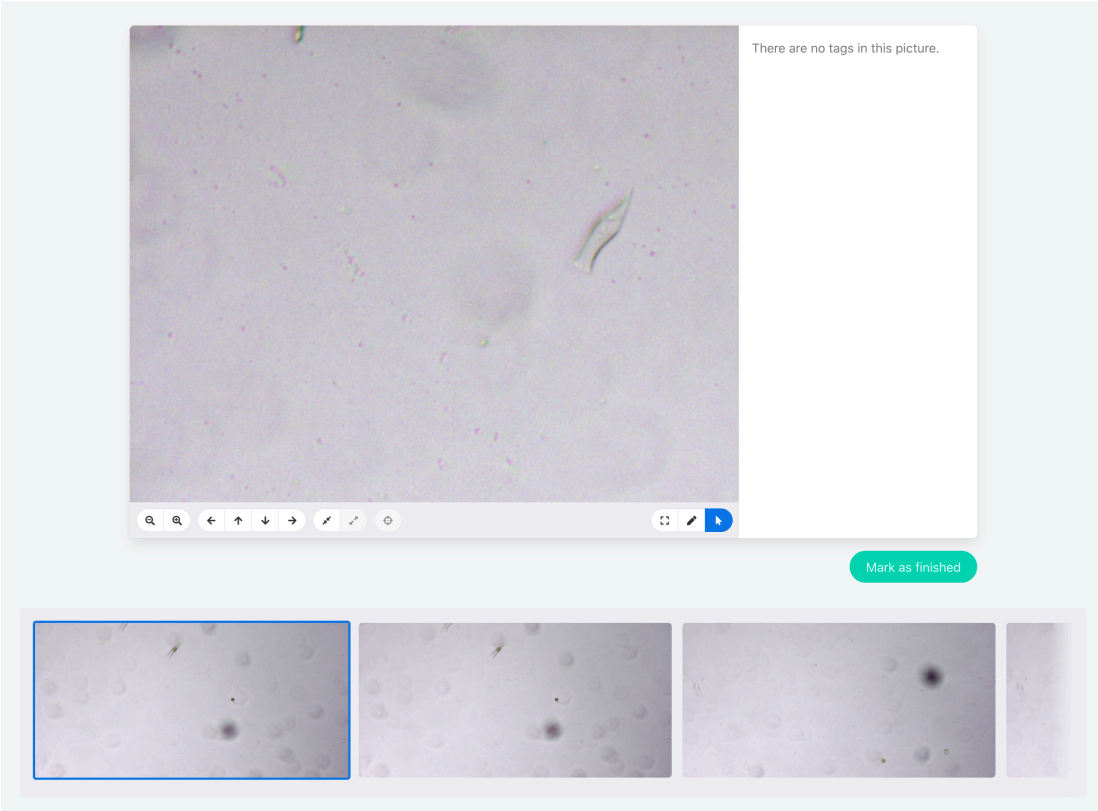

Figure 6.13: The assignment view that shows the Boxer UI.

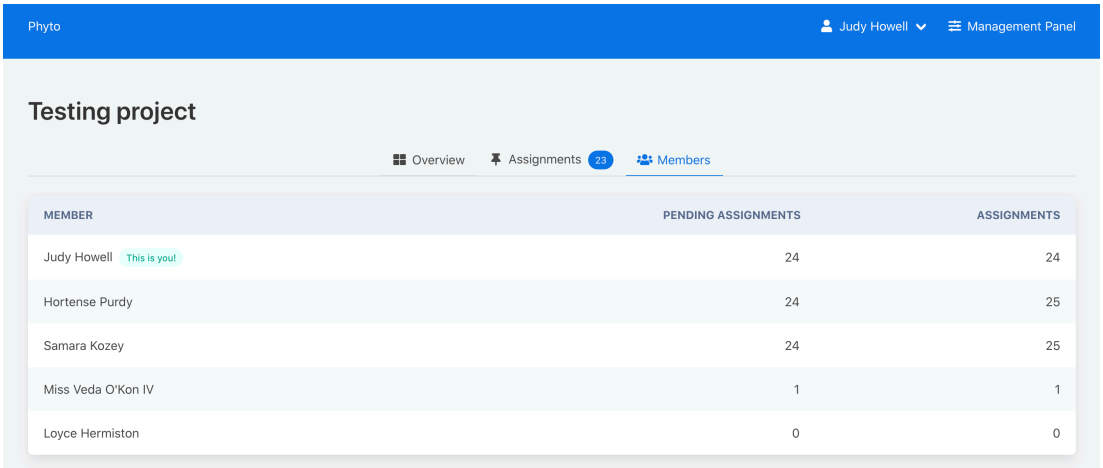

Figure 6.14: The member list of a project.

Chapter 7
Conclusions

Finally, we will be dedicating the following lines to point out everything we can learn from the development of this project.

7.1 Conclusions

After evaluating the necessities of a group of scientists, we have provided a tool from which they can base their work, aiding them in their daily tasks while also opening the door to automate part of it. With this platform, they can now work on their identification projects while having all of their data safely stored in one place.

The user interface of our application was built to be easy to use so that any of those biologists can engage into the platform with ease, to make them focus on their work, and nothing else.

We also made sure that we provide a performant solution that allows to easily work with data over the internet that can potentially be worth a lot of space, squeezing the last byte we have thanks to our image compression system.

Finally, we created a way to interact with images that is easy and fun to do, while keeping track of all the tasks each individual worker must do, giving them only one place where they can look their pending work.

7.2 Methodology

The methodology used, Scrum, brought to us the perks and disadvantages of agile development. First, we learned that the iterative nature of scrum helped to obtain a more consistent product, thanks to the refinements done in every sprint and to the meetings with the product owner.

Secondly, we can learn from this project that for the Story Points estimation approach to

work, the development team must be very experienced in similar tasks to ensure the maximum precision.

Finally, we also learned that deviations are part of our field and underestimations happen, so we must also be prepared and know how to act when they do.

Among all the positive factors Scrum can bring to a project, it is worth mentioning the closeness of the team. By having constant meetings and being the Product Owner and the Development Team so close to one another, it created a great atmosphere among everyone involved, were new features and solutions were imagined in inspiring brainstorming sessions. Thus, making the development team feel close to the product at hand and part of the common objectives.

7.3 Future Work

It is true that while the main features imagined for this platform were implemented, a lot of great ideas to improve the quality of life were left behind. Here are some improvements that could be done to the platform itself and to the project.

7.3.1 Taxonomy View

Most of species names in the taxonomy database are just common scientific names. We could use those names and some great resources out in the internet, to provide thumbnails of every species, alongside some imaging to easily recognize how they would look and, thus, providing some in-place information for biologists to know whether their identification resembles reality.

7.3.2 Project cloning

Oftentimes projects are based or very similar to one another. By providing a tool that allows project managers to base a configuration of a new project from another, we could not only help when starting projects but also we could use it as an opportunity to relate functionality between similar projects.

7.3.3 Continuous Integration

Unfortunately, most of the analysis tools we used were run locally. Mainly because the development team was made up by one developer. It would be great to set up a server that would take care of the analysis of every commit done.

7.3.4 Deploy

Our platform was never actually deployed, so we could never test its performance in real life. Doing some performance testing under the deployment environment could lead to great improvements. Moreover, by being used, we could obtain feedback from the actual biologists to tailor the application even more to their needs.

Finally, the deployment is a great opportunity to set up some nice DevOps utilities like containers and zero downtime deployments.

7.3.5 Automated Services

Automated services that could be used to integrate with our platform already exist, but there are not many. Creating new services, specialized in species from common projects should be our main goal for the future.

In case of the already existing automated services, it is crucial to create a REST interface so that we can communicate our platform with them.

Appendices

Appendix A

Setup

The following lines will try to explain how to successfully set up a development environment of the application as well as how to correctly configure the application in a production environment.

A.1 Requirements

Independently of whether you want to set up a development or a production environment, the following requirements must be held:

- PHP 7.4 or greater.
- Node.js 14.4 or greater (with npm).
- Composer 1.10 or greater.
- Cron
- MySQL 5.6+ (recommended SQLite 3.8.8+ instead for development environments).

You will also need the following PHP extensions to be installed:

- php-common
- php-bcmath
- php-openssl
- php-json
- php-mbstring
- php-imagick
- php-zip

A.2 Development environment

To install the development environment we will be assuming that a UNIX-like system is being used.

First, place a copy of the project somewhere in your computer. Throughout these instructions will be assuming the `project/` path to be the project main directory.

With the terminal already in our project's main directory (`cd project/`), let's create a file to hold our SQLite database. We can do this inside the `project/database/` directory:

```
touch database/database.sqlite
```

Next up, we will install our projects dependencies. This should take a while, don't worry about it.

```
composer install
```

Once that is finished, we will duplicate the file `project/.env.example` and name the new copy `.env`. This file will hold all of our environment variables to configure the project.

```
cp .env.example .env
```

Now edit the newly created `.env` file with your editor of choice:

```
vim .env
```

Most of the default variables would suffice for a normal use of the application. However, we will be needing to modify a few of them to set the database. As we are using an SQLite database, change the DB_CONNECTION variable to *sqlite* and the DB_DATABASE variable to the **absolute** path of the database file. Then, you can optionally remove all other DB_ variables as they will not be necessary anymore.

Ultimatelly, you should have something like this:

```
DB_CONNECTION=sqlite
DB_DATABASE=/absolute/path/to/the/project/database/database.sqlite
```

Now we will generate an app key for our application to be able to securely encrypt cookies and prevent cookie poisoning. This is automatically done through Laravel's Artisan toolkit:

```
php artisan key:generate
```

It is time for us to migrate our database with all the application tables. As always, thanks to Laravel this is very easy:

```
php artisan migrate
```

We can also populate (seed) the database with some dummy data to work with:

```
php artisan db:seed
```

We also have some special seeds that will create an admin account with `phyto@phyto.test` as the email and `admin` as the password, so we can directly log in into the application. Remember that the application is private and, thus, is not possible to register without permission of an admin.

```
php artisan db:dev
```

It is now the turn to compile all the front end assets. For this we will first need to install all frontend related dependencies. Because we are using node to manage all of the frontend files, we need to use `npm`:

```
npm install
```

Following the previous command we are now able to compile the assets. Because we are setting up a development environment, we will compile assets for this specific environment, which does not minify the code and also creates map files, for easy local debugging.

```
npm run dev
```

We can now create a server for our application like this:

```
php artisan serve
```

This will create a local server (usually on port 8080) that we can access though our favorite browser in `http://localhost:8080`.

Please take into account that this server is meant for testing purposes. If you want to consistently develop the application you would want to use something like a LAMP environment or Laravel Valet.

A.3 Mailing setup

It is worth noting that the application will need for you to configure a mailing driver that will be in charge of the delivery of all mails that the application will send.

To configure the mailing driver you would like to use, please refer to Laravel's documentation on Mail [1]. There's many options to choose from. For a development environment you'd likely want to use Mailtrap.

[1] https://laravel.com/docs/6.x/mail

A.4 Queue initialization

The application makes use of multiple queues to perform time consuming tasks in parallel, to prevent them from holding the time of response of a request.

To start the default queue process you will need to use the following command:

```
1 php artisan queue:work
```

Please note that the previous command will only start the queue in charge of mailing and other general purpose tasks of the application. You will also need to start another process for the queue that is in charge of image compression:

```
1 php artisan queue:work --queue=image-processing
```

You can run the following command to know more about `queue:work`:

```
1 php artisan help queue:work
```

A.5 Production configuration

We will be covering here what configurations must be done to the application to optimize it for it to be production ready.

First, optimize the autoloader files that are generated by composer:

```
1 composer install --optimize-autoloader --no-dev
```

Then cache all app configuration, all routes and views so that they are not retrieved every request.

```
1 php artisan config:cache
2 php artisan route:cache
3 php artisan view:cache
```

A.5.1 Cron job

It is also necessary to add the following cron job to take care of the scheduled tasks (for example, the garbage collection of orphan files).

```
1 * * * * * cd /path/to/project && php artisan schedule:run >>
    /dev/null 2>&1
```

A.5.2 Recomendations

Here are some production recommendations that you can take into consideration:

- Depending on the traffic of the application and its usage, you might want to create multiple processes for each of the queues.
- In order to get the most performance out of PHP, you should configure PHP-FPM in nginx to avoid creating a new php process for every request.
- Use a process monitor like Supervisor to ensure that the queue processes always stay running, even when a server might unexpectedly restart.

List of Acronyms

AR Active Record. 14

CSS Cascading Style Sheets. 15

DBMS Database Management System. 14

DOM Document Object Model. 15

HUD Heads-up Display. 50

JS JavaScript. 15, 50

LTS Long-term Support. 14, 28

ORM Object-relational mapping. 14

OS Operating System. 43

PB Product Backlog. 8–10, 19, 23, 49

PHP PHP: Hypertext Preprocessor. 13, 14, 16

PSR PHP Standards Recommendations. 13

SPA Single Page Application. 15

TLD Top Level Domain. 18

UI User Interface. 15, 23, 39, 40

UX User Experience. 23

Bibliography

[1] Thurman and Trujillo, *Outlines & Highlights For Introductory Oceanography*. Academic Internet Pub Inc, 2007.

[2] What is php? [Online]. Available: https://www.php.net/manual/en/intro-whatis.php

[3] Function arguments. [Online]. Available: https://www.php.net/manual/en/functions.arguments.php#functions.arguments.type-declaration.strict

[4] Eloquent orm documentation. [Online]. Available: https://laravel.com/docs/6.x/eloquent#introduction

[5] D. R. Martin Fowler, *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2003.

[6] Blade templates documentation. [Online]. Available: https://laravel.com/docs/6.x/blade

[7] Laravel valet documentation. [Online]. Available: https://laravel.com/docs/6.x/valet